AD-A035 178   OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC   F/G 9/2
THE ARCHITECTURE OF A DATABASE COMPUTER. PART II. THE DESIGN OF--ETC(U)
OCT 76   D K HSIAO, K KANNAN   N00014-75-C-0573
UNCLASSIFIED   OSU-CISRC-TR-76-2   NL

1 OF 2
AD
A035178

12

B.S.

TECHNICAL REPORT SERIES

COMPUTER & INFORMATION SCIENCE RESEARCH CENTER

THE OHIO STATE UNIVERSITY COLUMBUS, OHIO

12

(OSU-CISRC-TR-76-2)

THE ARCHITECTURE OF A DATABASE COMPUTER
PART II: THE DESIGN OF STRUCTURE MEMORY AND ITS RELATED PROCESSORS

by

David K. Hsiao and Krishnamurthi Kannan

Work performed under
Contract N00014-75-C-0573
Office of Naval Research

Computer and Information Science Research Center
The Ohio State University
Columbus, Ohio 43210
October 1976

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
OSU-CISRC-76-2 ✓

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER
9

4. TITLE (and Subtitle)
6 "The Architecture of a Database Computer, Part II: The Design of Structure Memory and its Related Processors,"

5. TYPE OF REPORT & PERIOD COVERED
Technical Report

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
10 David K./Hsiao
Krishnamurthi/Kannan

8. CONTRACT OR GRANT NUMBER(s)
15 N00014-75-C-0573 ✓

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Office of Naval Research
Information Systems Program
Washington, D.C. 20360

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
4115-A1

11. CONTROLLING OFFICE NAME AND ADDRESS

12. REPORT DATE
11 Oct[redacted], [redacted]76

13. NUMBER OF PAGES
110

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
14 OSU-CISRC-TR-76-2

15. SECURITY CLASS. (of this report)
12 113p.

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Scientific Officer	DDC New York Area
ONR BRO	ONR 437
ACO	ONR, Boston
NRL 2627	ONR, Chicago
ONR 102IP	ONR, Pasadena

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

10 to the 9th power

10 to the 10th power

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Database computer; security; clustering, partitioned content addressable memory; security atom; name mapping; structure memory; microprocessor; functional specialization; keyword transformation unit; structure memory; structure memory information processor; index transformation unit.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The database computer (DBC) is a specialized back-end computer which is capable of managing data of 10^9-10^{10} bytes in size and supporting known data models such as relational, network, hierarchical and attribute-based models. In addition to its intended purpose of handling large databases and interfacing with various data models, the DBC is one of the first database machines which have built-in protection mechanisms for access control and clustering mechanisms for performance enhancement.

407586

4/B

In this report, we present the design details of four components of the DBC. These four components form the structure loop of the DBC. They are the keyword transformation unit (KXU), the structure memory (SM), the structure memory information processor (SMIP) and the index translation unit (IXU). The KXU converts keywords into their internal representations. The need for and the implications of internal representations are discussed first. Data structures maintained and algorithms executed by the KXU are then presented. Finally, a hardware organization to realize the KXU is proposed.

The primary function of the SM is to retrieve and update structural information of the database. This information is likely to be large (10^7 -10^9 bytes). Furthermore, the operations on this information must be performed at a rate commensurate with that of database operations performed by the mass memory (MM). In this report, the concept of a partitioned content addressable memory (PCAM) is used to implement the SM with the above properties. Powerful PCAM organizations are possible using emerging technologies. To this end, three design alternatives using different technologies are examined. The three technologies are magnetic bubble memories, charge-coupled devices (CCDs) and electron beam addressable memories (EBAMs). Algorithms which manipulate the storage systems and auxiliary data structures maintained by the processing elements are also given. In order to enhance performance of the SM during heavy updates, a look-aside buffer memory is proposed.

The SMIP is responsible for performing set intersections on structural information retrieved by the SM. The concept of PCAMs is once again utilized to perform rapid intersection. The IXU is intended to decode the structural information output by the SMIP.

The four components are designed to operate concurrently. Keywords are sent to the KXU at regular intervals by the DBC's command and control processor (DBCCP). The output of KXU is sent to the SM which retrieves index terms for the transformed keyword predicates and sends them to the SMIP. The SMIP output is interpreted by the IXU and sent to the DBCCP. This pipeline of processors results in maximum utilization of the hardware.

The remaining components of the DBC form the data loop and are described in Part III.

PREFACE

This work was supported by Contract N00014-75-C-0573 from the Office of Naval Research to Dr. David K. Hsiao, Associate Professor of Computer and Information Science, and conducted at the Computer and Information Science Research Center of The Ohio State University. The Computer and Information Science Research Center of The Ohio State University is an interdisciplinary research organization which consists of the staff, graduate students, and faculty of many University departments and laboratories. This report is based on research accomplished in cooperation with the Department of Computer and Information Science. The research contract was administered and monitored by The Ohio State University Research Foundation.

TABLE OF CONTENTS

Page

ABSTRACT 1

1. INTRODUCTION 1

1.1 Background 1

1.2 Scope 1

2. THE KEYWORD TRANSFORMATION UNIT (KXU) 4

2.1 Keyword Transformation 4

2.2 Functional Specialization 6

2.3 The Logical Design of the KXU 6

2.3.1 Data Formats and Structures 7

2.3.2 Processor Logic 11

2.4 The Physical Realization of the KXU 13

2.4.1 Table Memories in the KXU 14

2.4.2 Implementation of KXU Logic 16

2.5 Implications of the KXU Design 16

3. THE STRUCTURE MEMORY (SM) 19

3.1 Performance Requirements and Logical Organization of the SM 19

3.1.1 Storage Consideration of the Directory Entry 22

3.1.2 The Role of Look-aside Buffer 22

3.2 Physical Realization of the SM 26

3.2.1 Data Structures for the Bucket Memory System (BMS) 26

3.2.2 Technology Based Design Alternatives for the BMS Storage 30

A. Magnetic-Bubble-Memory-Based Bucket Memory 30

B. Charge-Coupled-Device-Based Bucket Memory 41

C. Electron-Beam-Addressable-Memory-Based Bucket Memory 52

D. BMS Designs in Retrospect 56

3.2.3 The Processing Elements (PEs) of BMS 57

A. Number and Nature of the PE 57

B. Data Structures Maintained by the PE 58

C. Size and Load Conditions of the Data Structures 63

D. Logic of the PE 68

E. PEs and Structure Memory Controller 71

3.3 The Structure Memory Controller (SMC) 72

3.3.1 Transformation of Logical Bucket Name to Physical Bucket 72

3.3.2 The DBCCP Commands Executed by the SM 74

Page

3.3.3 Relationship of SMC, SMIP, DBCCP and KXU 77

3.4 The Look-Aside Buffer Memory (LABM) 79

3.4.1 Design and Implementation of the LABM 79

3.4.2 Effects of the LABM on SMC Command Execution 80

4. THE STRUCTURE MEMORY INFORMATION PROCESSOR (SMIP) 84

4.1 Logical Description of the SMIP 84

4.2 Implementation Considerations 87

4.3 Physical Organization of the SMIP 90

4.3.1 The SMIP Controller 90

4.3.2 The Processing Element and Memory Unit Pairs 94

5. THE INDEX TRANSLATION UNIT (IXU) 98

5.1 The Need for an IXU 98

5.2 Data Structures in the IXU 99

5.3 Algorithms Executed by the IXU 99

5.4 Hardware Considerations 103

6. CONCLUDING REMARKS 105

REFERENCES 106

1. INTRODUCTION

1.1 Background

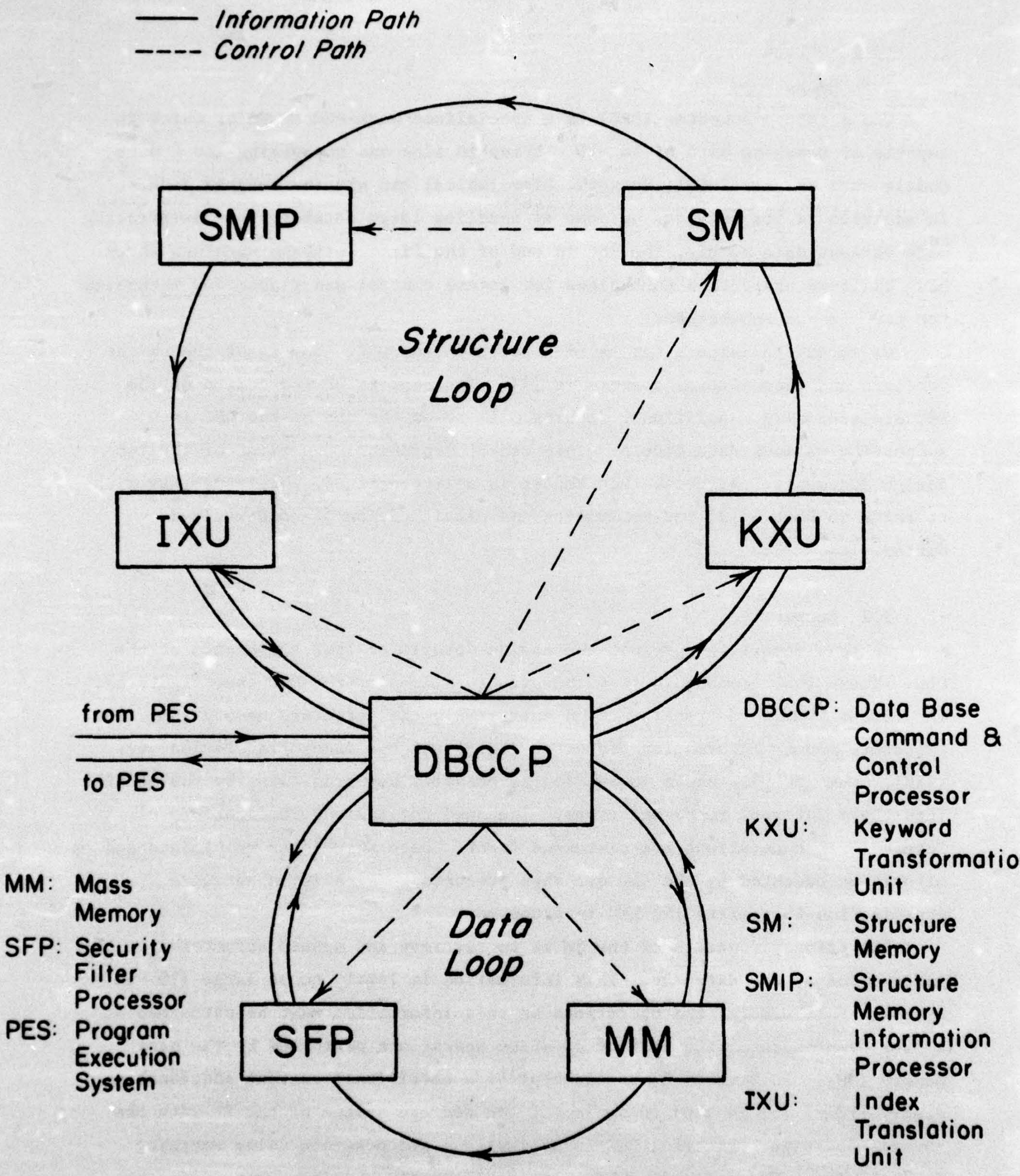
The database computer (DBC) is a specialized back-end computer which is capable of managing data of 10^9-10^{10} bytes in size and supporting known data models such as relational, network, hierarchical and attribute-based models. In addition to its intended purpose of handling large databases and interfacing with various data models, the DBC is one of the first database machines which have built-in protection mechanisms for access control and clustering mechanisms for performance enhancement.

We intend to issue a series of reports on the DBC. The first one on the concepts and capabilities appears in [3]. Two reports on the design of the DBC are scheduled. Additional reports will be on the use of the DBC in supporting various data models. This report represents the first of the two design documents. Although this report is self-contained, the reader may wish to refer to Part I [3] for motivation and clarification of concepts and definitions.

1.2 Scope

In this report, we present the design details of four components of the DBC. These four components form the <u>structure loop</u> of the DBC (see Figure 1). They are the keyword transformation unit (KXU), the structure memory (SM), the structure memory information processor (SMIP) and the index translation unit (IXU). The KXU, discussed in Section 2, converts keywords (sent by the DBCCP) into their internal representations. The need for and the implications of internal representations are discussed first. Data structures maintained and algorithms executed by the KXU are then presented. Finally, a hardware organization to realize the KXU is proposed.

The primary function of the SM is to retrieve and update structural information of the database. This information is likely to be large (10^7-10^9 bytes). Furthermore, the operations on this information must be performed at a rate commensurate with that of database operations performed by the mass memory (MM). In Section 3, the concept of a partitioned content addressable memory (PCAM) [3] is used to implement the storage system of the SM with the above properties. Powerful PCAM organizations are possible using emerging technologies. To this end, three design alternatives using three different technologies are examined. The three technologies are magnetic bubble memories, charge-coupled devices (CCDs) and electron beam addressable memories (EBAMs). Algorithms which manipulate the storage systems and auxiliary data



Information Path
Control Path
SMIP
SM
Structure Loop
IXU
KXU
from PES
to PES
DBCCP
Data Loop
SFP
MM
DBCCP: Data Base Command & Control Processor
KXU: Keyword Transformation Unit
SM: Structure Memory
SMIP: Structure Memory Information Processor
IXU: Index Translation Unit
MM: Mass Memory
SFP: Security Filter Processor
PES: Program Execution System

Figure 1. Architecture of DBC

structures maintained by the processing elements are also given. In order to enhance performance of the SM during heavy updates, a look-aside buffer memory is proposed.

The SMIP is responsible for performing set intersections on structural information retrieved by the SM. In Section 4, the concept of PCAMs is once again utilized to perform rapid intersection. The IXU is treated in the fifth section and is intended to decode the structural information output by the SMIP.

The four components are designed to operate concurrently. Keywords are sent to the KXU at regular intervals by the DBCCP. The output of the KXU is sent to the SM which retrieves index terms [3] for the transformed keyword predicates and sends them to the SMIP. The SMIP output is interpreted by the IXU and sent to the DBCCP. This pipeline of processors results in maximum utilization of the hardware.

The remaining components of the DBC form the data loop (see Figure 1) and are described in Part III [8].

2. THE KEYWORD TRANSFORMATION UNIT (KXU)

The keyword transformation unit (KXU) is intended for the performance enhancement of the structure memory (SM) in the search, storage, and processing of directory entries. The manner in which keywords are represented in the SM plays a pivotal role in the efficient execution of these tasks. Thus, it is important to examine the issues involved in the internal representation of keywords and the mechanism employed to carry out the transformation of external keywords into their internal representation. In this section, we propose a hardware organization to realize keyword transformation.

2.1 Keyword Transformation

The most frequently used SM operation is the search command which retrieves directory entries of those keywords which satisfy the predicate associated with the search command. One of the purposes of transforming keywords is to enable the SM to readily identify the sectors of the partitioned content addressable memory (PCAM) to be searched for a given keyword. (Recall from [3] that the SM is implemented using a PCAM). Hashing as a search technique accomplishes this. The basic strategy of any hashing technique is to partition the search space into mutually exclusive sets called buckets. Buckets usually comprise one or more sectors of the PCAM. Such an arrangement, then, allows the SM to search for the directory entry of a keyword within the confines of a few sectors instead of the entire PCAM.

Another reason for transforming keywords is related to efficient use of PCAM storage. Keywords used by the database user are not constrained to be of fixed length. Variable-length keywords will, in general, require larger storage space than fixed-length ones, since a variable-length keyword will require a field to indicate the length of the keyword in addition to the keyword itself. Thus, it is advantageous to store keywords in their encoded form of fixed length.

A third reason is related to the efficient processing of information which is known a priori to be of fixed length. Since variable-length keywords are internally represented as fixed-length fields, keyword comparisons can be made at a faster rate in the SM.

From the discussion, we conclude that the keyword transformation involves the encoding of variable-length keywords into fixed-length form by hashing.

How does the use of hash encoded directory entries affect the operation of the DBC? In a directory where keywords are stored in their original form, all directory entries are distinct. However, this is not guaranteed to be the case when transformed keywords are used, since two keywords could be transformed into the same hash code (i.e., there is a collision). We will call this structural ambiguity. In order to determine the severity of structural ambiguity existing in the DBC, we shall consider the probability p of one or more keywords being transformed into the same hash code (i.e., the probability of one or more collisions).

$$p= 1-[n(n-1)\ldots(n-(N-1))]/n^N$$

where N is the number of keywords and n is the address space determined by the length L of the hash code.

The inability of the SM to distinguish between keywords having the same internal representation may result in MAU accesses which do not have any record satisfying a user query. (For a given MAU address f, the MAU access is defined to be the process of locating and searching the f-th MAU by the mass memory (MM). Note that this does not imply that records in such MAUs will be passed on to the user). We shall call this problem arising from structural ambiguity, access imprecision. How often does the problem of access imprecision take place in the system? To answer this question we need to compute the probability p' that a partition of the MM will be searched for a query Q even though all of the keywords *in the query are not* in the keyword basis of the partition. Since a partition of the MM is characterized by a triple (MAU address f, cluster number c, security atom name s), the keyword basis of the partition is

$$KB(f,c,s) = \{K|\exists R \,\varepsilon\, (f,c,s) \ni K \varepsilon R\}.$$

Let the cardinality of KB(f,c,s) be m. Further, let us assume that the query Q is a conjunct of n equality keyword predicates and that r out of n of these keywords also appear in KB(f,c,s) for some f, c, and s. Then we can show that

$$p' < (\frac{nm}{2^L})^{n-r}.$$

When all keywords of Q except one appear in KB(f,c,s) (i.e., n-r = 1), then

$$p' < \frac{nm}{2^L}$$

Under normal circumstances, nm << 2^L, so p' will be quite small. For example, for nm = 1,000 and L = 48, p < 4 × 10^{-12}; for nm = 1,000 and L = 32, p' < 2.5 × 10^{-7}; and for nm = 1,000 and L = 24, p' < 6.2 × 10^{-5}. When more than one keyword of Q is not in KB(f,c,s), then the value of p' becomes vanishingly small. Furthermore, we observe that L can always be chosen to make p and p' as small as we please. Thus, we conclude that the problem of access imprecision due to structural ambiguity will not affect the performance of the DBC in any significant manner.

2.2 Functional Specialization

Since every keyword has to undergo transformation before the SM is interrogated, one might argue for the integration of the transformation logic with that of the SM itself. However, there are strong reasons for identifying the transformation logic as a separate functional unit.

First, the operation of the SM is functionally different from that of the KXU. Searching on fixed-length fields is the single most important activity taking place in the SM. In hardware terms, it involves efficient comparison operations. The KXU, on the other hand, manipulates variable-length fields, and includes shifting and rotating of the characters of keywords, arithmetic operations on (arrays of) characters, and indexing for loop control. The KXU should, therefore, be equipped with general-purpose and indexing registers on which the above type of operations can be carried out.

Second, the SM and the KXU are meant to operate in parallel. That is, as soon as a keyword has been transformed, the KXU is ready to process the next keyword sent by the DBCCP. Meanwhile, the SM is retrieving, deleting, or inserting the index term(s) for the transformed keyword. Such parallelism is difficult to achieve if the SM is required to perform both keyword transformation and index term retrieval or update.

Third, modularity is achieved by clearly identifying and separating the functions of the KXU and the SM. It also lends itself to clear and simple interfaces between the components involved.

Finally, the incorporation of reliability features is facilitated by considering the reliability requirements of each of the components separately. The SM is mainly a storage system requiring some form of redundancy coding to enhance reliability, while the KXU is essentially a computing device requiring redundancy in the processor to enhance reliability (e.g., triple modular redundancy [12]).

2.3 The Logical Design of the KXU

Good keyword transformation algorithms are highly dependent on the nature of the keywords and their expected use. It is unreasonable to expect a single transformation algorithm to be equally applicable to all keywords because keywords of different files could have widely varying properties and could appear in different types of queries. For example, one file could be queried on the basis of simple conjunct of equality predicates, whereas the queries for another file might consist of the less-than predicates. Further, we would require that directory entries of keywords belonging to different files, be stored in different parts of the SM. Since storage in the SM is based on the buckets associated with keywords, (see

Section 3), the KXU must ensure that keywords of different files are associated with different buckets. The above discussions lead us to the following data structures and processor logic.

2.3.1 Data Formats and Structures

Each file that is known to the system has a unique identification number (ID). The DBCCP, in all its requests to the KXU, sends two pieces of information - the ID of the file to which the keyword belongs and the keyword proper. This is depicted in Figure 2. The keyword, in turn has two parts: an attribute identifier and a value field. This is shown in Figure 3. The attribute identifier identifies the attribute of the keyword uniquely, not only among distinct attributes within the same file but also among identical attributes of different files.

A transformed keyword T(K) consists of two parts: a 24-bit logical bucket name and a 24-bit encoded representation of the keyword value. As shown in Figure 4, the logical bucket name consists of two parts: a 16-bit attribute id (which is identical to the one supplied by the DBCCP), and an 8-bit partition number. The 256 partitions, that can be identified by the 8-bit partition field are used for categorizing keywords of an attribute according to their values. For example, if the value of an attribute 'salary' ranges from say $5,000 to $100,000, then we might partition this range into 256 equal subranges. Thus, given a keyword, say, salary = 50,000, we can easily determine its partition number as follows:

$$\text{Partition Number} = \left\lfloor \frac{50{,}000 - 5{,}000}{(100{,}000 - 5{,}000)/256} \right\rfloor$$

In case the attribute of a keyword hes a non-numeric value (i.e., alphanumeric value), partitioning of the values is done on the basis of the encoded order of characters. One of the nice properties of the partitioning scheme is that it preserves natural ordering among keywords of an attribute. This leads us to observe that partitions are created to facilitate inequality predicates so that they can be processed in a straightforward way by the SM.

The remaining 24 bits of the transformed keyword are obtained by applying a hashing algorithm to the keyword value. Each active file in the system (i.e., a file that is currently being accessed) is allowed to have a maximum of four different

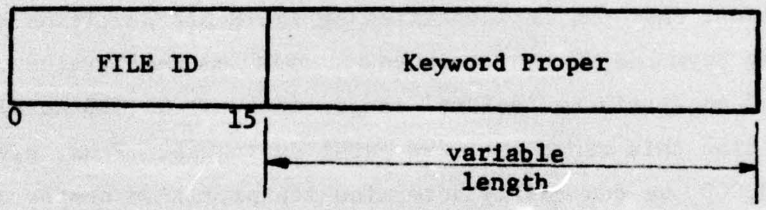


Figure 2. Input from DBCCP

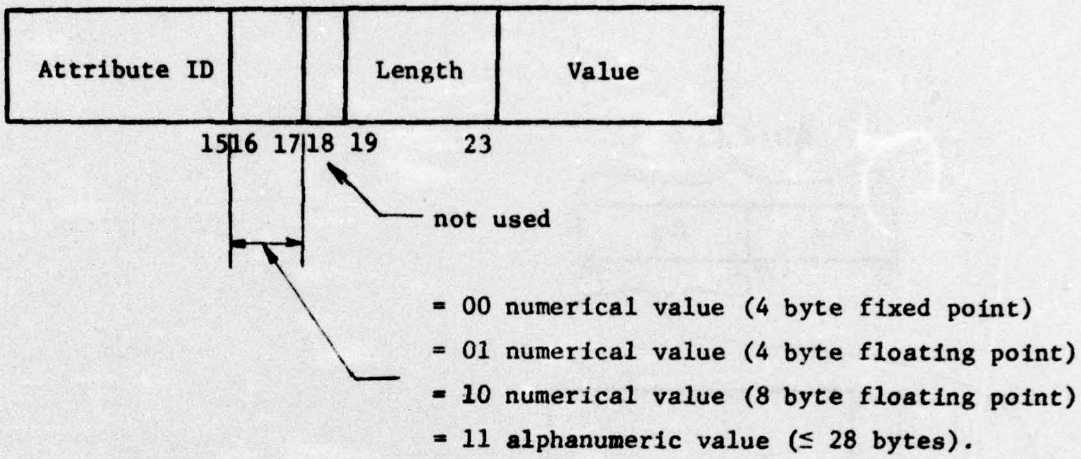

Attribute ID
Length
Value
15
16
17
18
19
23
not used
= 00 numerical value (4 byte fixed point)
= 01 numerical value (4 byte floating point)
= 10 numerical value (8 byte floating point)
= 11 alphanumeric value (≤ 28 bytes).

Figure 3. Format of a Keyword Proper

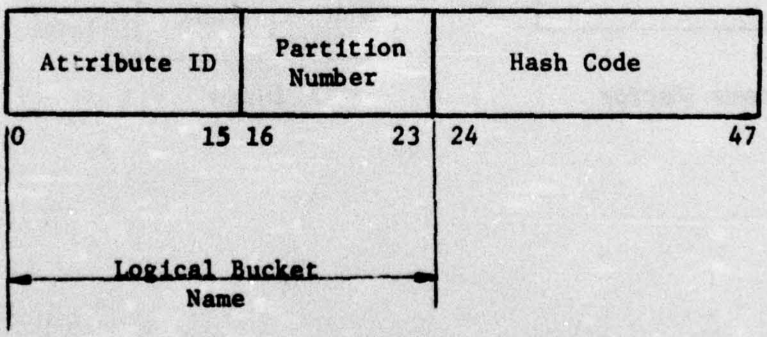


Figure 4. The Parts of the Transformed Keyword T(K)

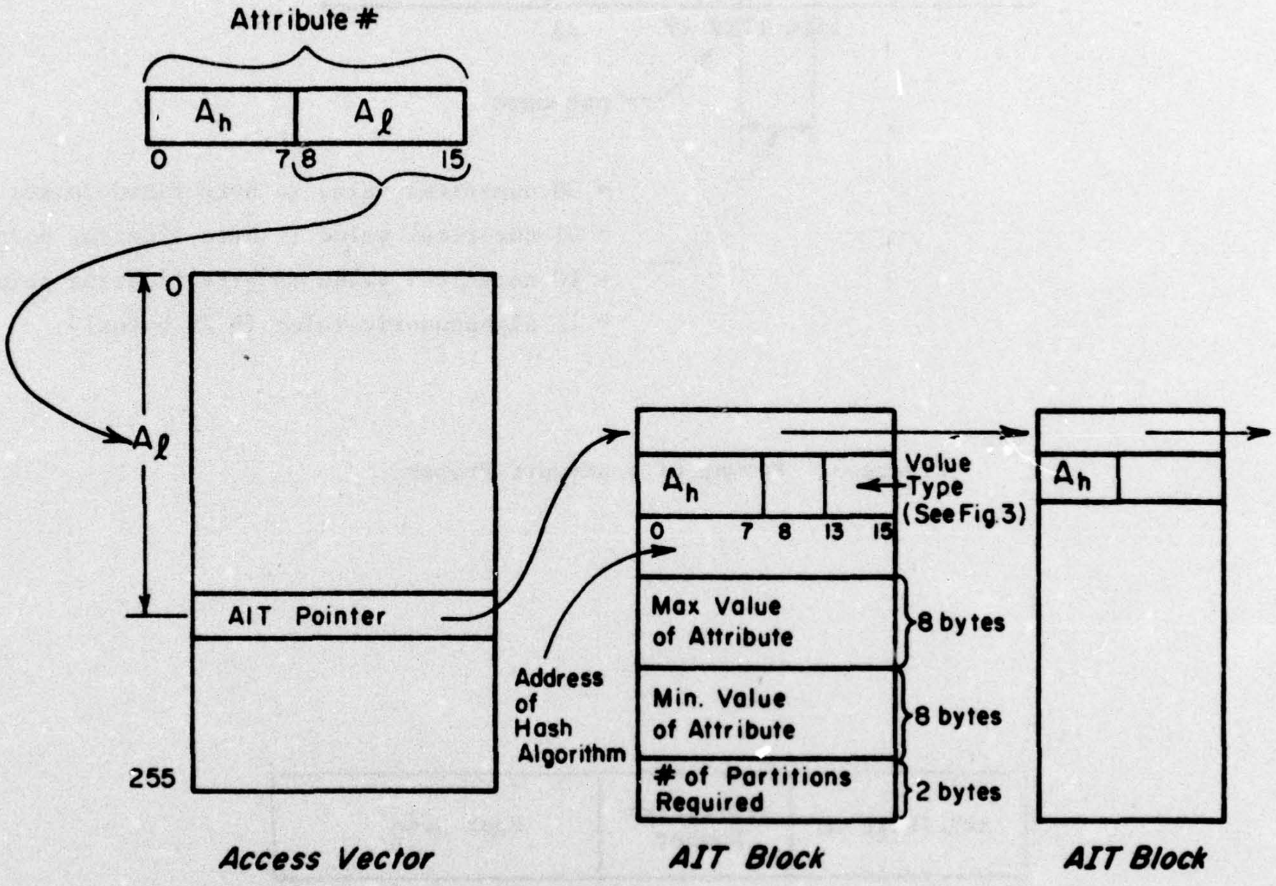


Figure 5. Attribute Information Table (AIT)

this information is accessed and stored within the table. The AIT has two parts: the access vector of 256 entries and a variable number of information blocks called AIT blocks. Given the attribute identifier, the access vector is used to access in a rapid fashion the information block of the attribute. Each 24-byte AIT block contains the following information:

(a) pointer to the next AIT block belonging to an attribute with same 8 low order bits,

(b) address of hash algorithm,

(c) 8 high order bits of attribute identifier,

(d) value type : 00 for fixed point number, 01 for short floating point number, 10 for long floating point number, and 11 for alphanumeric values,

(e) maximum value of attribute,

(f) minimum value of attribute and,

(g) number of partitions desired (≤ 256).

The AIT blocks are allocated dynamically as required.

Another important data structure maintained by the KXU is the hash algorithm library (HAL). This library contains the hash algorithms of all the active files in the DBC. The HAL is organized into blocks of 400 bytes. No hash algorithm is expected to be longer than 400 bytes. (This is a design decision which can be changed without affecting the logic or performance of the DBC). Thus, each hash algorithm is stored completely in one block. HAL memory is allocated (and freed) one block at a time. The hash algorithm addresses in AIT blocks (see Figure 5) point to blocks in the HAL memory.

2.3.2 Processer Logic

The KXU operates in two distinct modes. These modes are known as the load mode and the translate mode. By means of control signals the DBCCP can set the KXU to the desired mode of operation. In the load mode, the KXU accepts information regarding a file. Such information is used by the KXU to build the AIT. In the translate mode, the KXU performs keyword transformations. Algorithms presented here will be categorized as either load or translate type depending on whether they perform in load or translate mode.

ALGORITHM A (Load Type) - To build a set of hash algorithms for a file.

Input Arguments from the DBCCP: Hash algorithms on the input data lines. [Assume k (≤ 4) algorithms are to be read]

```
Step 1:  $i \leftarrow 1$.
Step 2:  Read the i-th hash algorithm into the input buffer.
Step 3:  Request a block in the HAL memory.  (See algorithm D for
         HAL allocations).  Let the address by $HAL_i$.
Step 4:  Move hash algorithm from input buffer into the block at $HAL_i$.
Step 5:  Set ADDR [i] to $HAL_i$.  [ADDR is an array of four address
         words which are used to remember the addresses of the hash
         algorithms of a file within HAL.  ADDR is subsequently used
         by algorithm B to set hash address pointers in the AIT blocks
         in step 9].
Step 6:  $i \leftarrow (i + 1)$.  If $i \leq k$, then go to step 2; else, terminate.
```

ALGORITHM B (Load Type) - To create a set of AIT blocks for a file.

Input Arguments from DBCCP: Attribute identifiers and related information. [Assume n AITs have to be created.]

```
Step 1:  $i \leftarrow 1$.
Step 2:  Read the i-th attribute identifier and the related information
         from the DBCCP into the input buffer.
Step 3:  Request an AIT block by using Algorithm C.
Step 4:  Use the 8 low order bits of the attribute identifier $A_\ell$ to
         address the access vector and let CURRENTPTR $\leftarrow A_\ell$.
Step 5:  Retrieve the AIT pointer pointed to be CURRENTPTR.  Call
         it NEXTPTR.
Step 6:  ACCESSVECTOR [CURRENTPTR] $\leftarrow$ Address of new AIT block.
Step 7:  Set AIT pointer in the new block to NEXTPTR.
Step 8:  Move information from input buffer into AIT block.  Using
         information in ADDR array (see step 5 in Algorithm A) set
         hash address pointer in new AIT block.
Step 9:  $i \leftarrow i + 1$.
Step 10: If $i \leq n$, then go to step 2; else, terminate.
```

ALGORITHM C (Load Type) - To allocate an AIT block. (In order to maintain AIT blocks, a bit map is used)

Input arguments - None.

```
Step 1:  Scan the bit map for the first bit which is 0.  If no bit is
         found, signal error condition.  Turn the bit on.
Step 2:  Compute the AIT block address corresponding to the bit found
         in step 1.  Terminate.
```

ALGORITHM D (Load Type) - To allocate a HAL block. (In order to maintain HAL blocks, a bit map is used).

Input Arguments: None.

```
Step 1:  Scan the bit map for the first bit which is 0.  If no bit is
         found, signal error condition.  Turn bit on.
```

Step 2: Compute the HAL block address corresponding to the bit found in step 1. Terminate.

ALGORITHM E (Load Type) - To deallocate a AIT/HAL block.

Input Arguments: None.

Step 1: Compute the bit number in the bit map corresponding to the block to be deallocated.
Step 2: Turn off the bit. Terminate.

ALGORITHM F (Load Type) - To delete the AITs and hash algorithms of a file.

Input Arguments (from DBCCP): m attribute identifiers.

Step 1: $i \leftarrow 1$.
Step 2: Use the 8 low order bits of the i-th argument attribute identifier as index into the access vector.
Step 3: Follow the pointer in the access vector entry until a match is obtained for the 8 high order bits of the argument attribute identifier.
Step 4: Release the HAL block pointed to by the hash algorithm address field of the AIT block found in step 3. (Use Algorithm E).
Step 5: Release the AIT block found in step 3. (Use Algorithm E).
Step 6: $i \leftarrow i + 1$: if $i \leq m$, go to step 2; else, terminate.

ALGORITHM G (Translate Type) - To transform a given keyword proper into its encoded form.

Input Arguments: Keyword supplied by the DBCCP.

Step 1: Use the 8 lower order bits of the attribute identifier in the keyword proper to index into the access vector.
Step 2: Follow the pointer in the access vector until the 8 high order bits match with the 8 bits stored in an AIT block.
Step 3: Retrieve the address of the hash algorithm from the AIT block and load the hash algorithm from the HAL into the control store.
Step 4: Compute partition number for the keyword using the partition details in the AIT block and the keyword value.
Step 5: Invoke the hash algorithm in the control store with the keyword value as argument.
Step 6: Form the 48-bit transformed keyword by concatenating 16 bits of attribute identifier, 8 bits of partition number computed in step 4, and 24 bits of hash code computed in step 5.
Step 7: Send the 48 bit transformed keyword to the SM and to the DBCCP. Terminate.

2.4 The Physical Realization of the KXU.

From the above discussion, we can estimate the type and amount of hardware required to realize the KXU. First, we need enough memory to hold the AIT and HAL. Second, we require a mechanism to execute the seven processor algorithms and the hash algorithms.

2.4.1 Table Memories in the KXU.

Since the AIT is frequently accessed (once for every keyword transformation), the performance of the KXU can be affected by the access time to the AIT. Furthermore, the organization of AIT requires at least two and possibly more accesses before the required AIT block can be retrieved. Thus a random access memory is best suited for implementing the AIT. The access times per word can be in the range of 0.5 to 1 μsec, since we need to access a small number (6) of words (= 24 bytes) at a time. We must now determine the size of the AIT memory. Since each attribute requires one AIT block and an AIT block occupies 24 bytes, it is easy to compute the AIT size, if the file characteristics (i.e., the number of attributes per file) and the number of active files in the system are given. In Table I, we have tabulated the AIT sizes in terms of the number of attributes and files. We next ask ourselves what technology we should use to implement AIT. There are a number of technologies which can provide the performance characteristics that we need, but we should choose the one with the lowest cost. The leading contenders for RAMs with access times of 0.5 - 1 μsec are core technology and MOS/LSI technologies. Core is more competitive beyond 200 Kbytes while MOS/LSI has the edge for sizes less than 200 Kbytes. This is illustrated in the table by the broken line.

We now consider the hash algorithm program library (HAL) memory. In Table II, we have tabulated the HAL size for different numbers of active files in the DBC and for different hash algorithm sizes. (Four hash algorithms are assumed for every active file.) The address of a hash algorithm within the HAL to be used for a keyword is determined by looking up the corresponding AIT block. Thus, the number of accesses to the HAL for the retrieval of a hash algorithm is exactly one. This implies that, besides random access memory, quasi-random access memories or even sequential access memories might be used for storing the HAL. Although random access memories can be used to implement the HAL, we can look for cheaper alternatives in fixed-head disks/drums and sequential CCD memories. Access time for fixed-head disks consists of only latency times which vary from 5-10 msec. Transfer rates can be as high as 1 Mbyte per second. Thus the total time to retrieve a hash algorithm from the HAL (implemented on fixed-head disks) will be in the 5-10 milliseconds range. CCD memories seem to be able to perform better for almost the same cost.

TABLE I: Size of AIT in Kbytes

Number of Active Files	Number of Attributes in File				
	25	50	75	100	150
50	30	60	90	120	180
100	60	120	180	240	360
150	90	180	270	360	540
200	120	240	360	480	720
250	150	300	450	600	900

TABLE II: Size of Hash Algorithm Library in Kbytes

Number of Active Files	Hash Algorithm Size in (bytes)				
	50	100	150	200	400
50	10	20	30	40	80
75	15	30	45	60	120
100	20	40	60	80	160
150	30	60	90	120	240
200	40	80	120	160	320
250	50	100	150	200	400

For example, clever CCD organization [6,11] can lower access times to the 10-100 μsec range. The transfer rate is as good or even better than fixed-head disks.

The final choice of technology for HAL depends on the access times required, since all the technologies discussed above can be used to build HAL memories of sizes tabulated in Table II. The access time to a hash algorithm is crucial to the overall operating speed of the KXU. This in turn is tied to the operating speed of the SM and other components in the loop (see Fig. 1). Thus, we have with us a number of technology alternatives with which to "tune" the KXU so that the KXU is never the performance bottleneck of the structure loop.

2.4.2 Implementation of the KXU logic

The KXU is proposed to be implemented using a microprogrammable LSI microprocessor. The processor's microprogram memory has two parts, a static store (implemented with a ROM) and a dynamic store (implemented with a writable control store). The static storage area contains the seven algorithms outlined earlier and a small control program to initiate the algorithms in proper sequence. The dynamic area contains the hash algorithm to be executed for a given keyword. There are several reasons for choosing a microprogrammable microprocessor. First, given the present state of the art of the semiconductor technology, it is far more economical to build control structures using array logic than random logic. Second, by using bit-slice microprocessor (like the INTEL 3000 series), decisions about data widths can be postponed until a very late stage in the design cycle. Third, the slower speed generally attributed to microprogrammable processors does not constitute a performance bottleneck in the DBC, since the SM component does not need to accept transformed keywords at rates faster than one every millisecond. Finally, the logic of the KXU is relatively straightforward so that the usual problems accompanying dynamic microprogramming (like protection of programs from one another) are absent.

In Figure 6, we have shown the organization of the various components that constitute the KXU.

2.5 Implications of the KXU Design

In the previous sections, we have presented a design for transforming the keywords of a file into an encoded form which can then be used by the SM

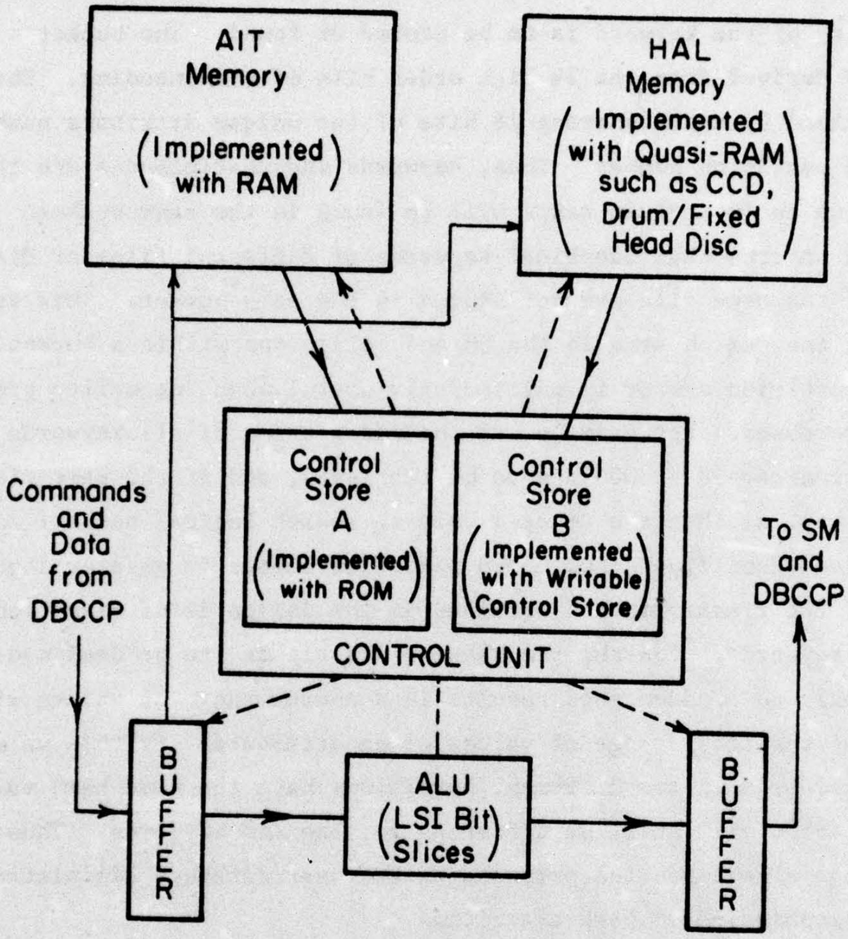

AIT Memory (Implemented with RAM)
HAL Memory Implemented with Quasi-RAM such as CCD, Drum/Fixed Head Disc
Control Store A (Implemented with ROM)
Control Store B (Implemented with Writable Control Store)
CONTROL UNIT
Commands and Data from DBCCP
To SM and DBCCP
BUFFER
ALU (LSI Bit Slices)
BUFFER

Figure 6. The KXU Organization

to store and retrieve directory entries. We now show that the KXU design indeed meets the goals established in the opening section.

First, the transformation uniquely identifies the bucket in which the directory entry of the keyword is to be stored or found. The bucket name in our design is derived from the 24 high order bits of the encoding. These 24 bits are obtained by concatenating 16 bits of the unique attribute number and 8 bits of the partition number. Thus, keywords whose attributes are the same and whose value is in a given range will be found in the same bucket. Such an arragement ensures that identical keywords of different files or different attributes of the same file are not stored in the same bucket. This results in minimizing the search area in the SM and collisions within a bucket. Second, the partition number is particularly useful when inequality predicates have to be processed. For example, if the index terms of all keywords K which satisfy the predicate K ≥ 3000 are to be retrieved, and if the partition number for 3000 is, say, n, then the SM need only to search logical buckets each of whose attribute identifies K and whose partition number is greater than or equal to n. Third, the transformation obtained in the design is of fixed length 48 bits for all keywords. Fourth, the hashing algorithms (to be designed by the users) need only to produce good results in a narrow range of values within a partition of the total range of values of an attribute. By this we mean that even if two keywords in two different partitions have the same hash value, the transformation will still be different for the two keywords. Thus, the proposed design alleviates the pressure on the user/database administrator to come up with sophisticated hash algorithms.

Finally, we make a few comments on reliability. The nature of the information held in the KXU is such that even a total loss of such information is not catastrophic. The AIT and the HAL are user supplied and thus can be re-constructed by requesting a reload from the PES. Standard parity check circuits may be employed with the memory systems to detect random errors and to request reloads. The processor logic is LSI-based and should require no special reliability feature.

3. THE STRUCTURE MEMORY (SM)

The structure memory (SM) is the repository of the directory entries of all the keywords known to the DBC. Retrieval operations on the mass memory (MM) are preceded by retrieval operations on the SM. Updates to the MM are accompanied by updates to the SM. Therefore, the performance requirements of the SM are dictated by a desire to ensure that this component does not become a bottleneck in the access path to the database residing in the MM. In meeting such performance requirements, we have proposed alternate designs - designs which use different emerging technologies. We have adopted this approach because it is not clear at the present time which, if any, of the emerging technologies will untimately become commercially viable. Another aspect of the designs presented in this section is that we have parameterized the designs to a great extent. The need to parameterize was prompted by our desire to offer the DBC as a viable alternative to users with a wide range of requirements. The DBC and in particular the SM should be capable of being tuned to a particular set of requirements by merely choosing the right set of design parameters.

3.1 Performance Requirements and Logical Organization of the SM

We begin our discussion of SM design by describing its performance characteristics. The SM must have sufficient capacity to store the directory entries for a database of 10^9 to 10^{10} bytes. Typically, directories are of the order of 1% to 10% of the database [7]. Therefore, the SM designs should be viable for capacities of 10^7 to 10^9 bytes. Furthermore, the SM must have sufficient speed to process queries at a rate commensurate with the speed of the MM. With good clustering strategy, a query will usually require one MAU access by the MM. Since the MM is implemented with modified moving-head disks and requires 15 to 25 milliseconds to access a MAU, the SM query processing time must be of the same order. Even though queries will vary widely in the number of predicates appearing in the queries, we estimate that in the worst case queries will seldom have more than 15 to 25 keyword predicates. Therefore, the SM must process each predicate in about 1 millisecond.

In order to meet the above requirement, the SM is organized as a partitioned content addressable memory (PCAM). A partition of the PCAM is defined to be the set of all of the directory entries that are accessed as a result of a single search order. In our discussion below, we have used the term logical bucket to denote a partition of the PCAM, and term bucket memory to denote the PCAM itself. These terms are historically associated with hashing; here they serve to remind the reader that partitions of the PCAM are identified by transformation of search keys.

Since transformed keywords are identified by their logical bucket names (see Fig. 4), the search for the directory entries of a keyword amounts to a search of the named logical bucket. By employing a large number of small logical buckets, we can reduce the amount of information to be searched in each one. The size of buckets, however, cannot be reduced indefinitely without incurring a performance degradation. The minimum bucket size is governed by the technology used to implement the SM.

In practice we cannot guarantee that the logical buckets will be equally filled. Therefore, buckets cannot be of fixed size. In conventional disk-oriented systems this problem is solved by allowing preallocated fixed-size buckets to spill over into shared overflow areas. There are two obvious drawbacks to this approach: First, the need for a shared overflow area increases the amount of data that must be searched. Second, by pre-allocating large fixed-size buckets, space wastage is inevitable. To avoid these problems we need to have truly variable-size buckets. The implementation of variable-size buckets can best be achieved by making the physical block size substantially smaller than the average bucket size.

The number of logical buckets is determined by the number of bits that are used to represent the bucket names. This, in turn, is dependent on the attribute name length and the partition name length. However, the actual number of buckets that can exist at any given instant is limited by the number of physical blocks that can be independently accessed by the SM. We thus make a distinction between logical buckets (created by the KXU) and physical buckets (that can actually exist) in the SM. This observation implies that the SM must employ a mapping device to maintain the relationship between logical bucket names and physical buckets.

The SM consists of three logical components: the bucket mapping unit, the bucket memory and the look-aside buffer (see Fig. 7). The bucket mapping unit is used to translate logical bucket names into physical bucket names. Conceptually, the bucket mapping unit contains a set of couples of the form (i,k) where i is the name of a logical bucket and k is the name of a physical bucket. The couple indicates that the logical bucket is stored in physical bucket k. The bucket memory maintains a variable number of variable-size

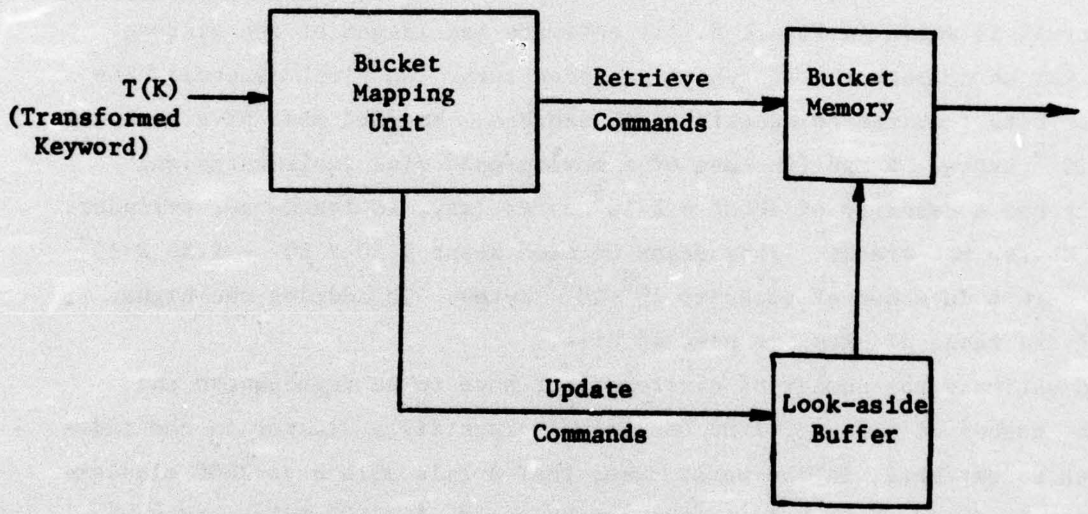


Figure 7. Logical Organization of SM

physical buckets. It accepts as input a physical bucket name and an access order, and operates by performing the indicated access on the physical bucket identified by the input physical bucket name. The look-aside buffer is used to buffer update operations in order to improve performance.

3.1.1 Storage Considerations of Directory Entries

As we recall from [3], a directory entry D(F,K) consists of a set of index terms

$$\{(f_1, c_1, s_1), (f_2, c_2, s_2), \ldots, (f_k, c_k, s_k)\}$$

where f is an MAU address, c is a cluster number, and s is a security atom name. Since a bucket cannot be shared by two or more files and since file names may be identified by logical bucket names, storage formats for index terms need not carry file names. Such a format, called the encoded directory entry format is shown in Figure 8. To estimate the length of the storage format, let us compute the length of an index term. We first determine the number of bits required to specify a MAU address. The DBC must have a capacity of 10^9-10^{10} bytes. A MAU (in case of a moving-head disk implementation) typically has a capacity of about 4 X 10^5 bytes (say, 20 tracks per cylinder, with 20 Kbytes per track). This means we need about 2.50 X 10^3 - 2.50 X 10^4 MAUs to hold a database of capacity 10^9-10^{10} bytes. To address the higher limit of the range of MAUs, we need 15 bits.

To estimate the number of clusters that have to be represented and hence the number of bits required to uniquely identify a cluster in the index term, let us estimate, in the worst case, that a file will have 1000 clusters. If the DBC is designed to hold in the neighborhood of 1,000 files, then the total number of clusters required to be represented is about 10^6. Thus we need 20 bits to represent a cluster name uniquely. The same kind of argument may be advanced to determine the number of bits required to uniquely represent a security atom name. We are now in a position to determine the length (in bits) of an index term. Each index term will require 55 bits (= 15 + 20 + 20). Can we do better than this? The answer, fortunately, turns out to be in the affirmative. Let us see how we can achieve a reduction in the index term size. First, recall from the previous section on the design of KXU that directory entries of keywords of different files will be stored in and retrieved from different logical buckets. Second, each query formulated by the user applies only to records within a particular file (i.e., a query cannot refer to more than one

T(K)	(f_1,c_1,s_1)	(f_2,c_2,s_2)	. . .	(f_k,c_k,s_k)

Figure 8. An Encoded Directory Entry

file). Third, no file is expected to occupy all of the MAUs. Fourth, no file is expected to have more than 1000 clusters or security atoms. These four observations imply a) that the index terms stored in the SM need not be unique, and b) the lengths (in bits) of each of the three index term components can be less than those required to represent the entire ranges of these components. Thus if a file is allocated to a maximum of n MAUs and has a maximum of p clusters and q security atoms, then an index term need only occupy $[\log_2 n] + [\log_2 p] + [\log_2 q]$ bits. The format of such an index term is shown in Figure 9. In the next section, we shall propose an index term format which reflects much of the above discussion.

What is the price that we have to pay in order to achieve this reduction? Since index terms are no longer unique, we need to have some kind of file dictionary which, given an index term, will produce MAU address, cluster number, and security atom number which can then be used to access the MM. We also need allocation and release mechanisms for MAU identifiers, cluster identifiers and security-atom identifiers. These are carried out by a functionally specialized component called the index translation unit (IXU) which is the subject of our discussion in a later section.

3.1.2 The Role of the Look-Aside Buffer

During the normal operations of the database, the retrieval of information from the SM will be far more frequent than the insertion (update) of (existing) information in the SM. However, it is conceivable that during short intervals of time, a large number of updates may have to be carried out. Such an occurrence can seriously affect the average information retrieval rate. The use of a look-aside buffer is aimed at alleviating such a degradation in SM performance. When an update or insert command is received by the SM, it is placed in the look-aside buffer. Execution of commands in the buffer is delayed until one of the following two conditions is met: (a) The loading of buffer reaches a certain level, called the threshold value; (b) the SM has no retrieve command awaiting execution. If and when any one of these conditions is met, the processing of the commands in buffer is taken up on a FIFO basis. In order to maintain a FIFO discipline, commands in the buffer are chained according to their arrival times.

The SM monitors the load (i.e., the number of entries) in the buffer in the following manner. When a new insert/delete command is received, it is placed at the bottom of the chain. The SM then checks the total number of

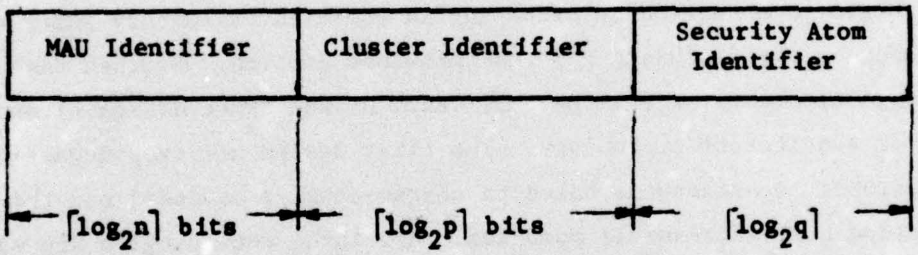


Figure 9. An Efficient Index Term Storage Format

entries in the buffer. If it is equal to or greater than threshold value, the SMC proceeds to execute all the commands from the beginning of the chain. The condition (b) is monitored as follows. After execution of a command, the SM sends a ready signal to the DBCCP. The SM then waits for a prespecified amount of time (called the time-out period) for another command from the DBCCP. If the time-out period expires without a command being issued by the DBCCP, the SM proceeds to execute the first command in the buffer.

3.2 The Physical Realization of the SM

The three logical components of the SM, (the bucket mapping unit, the bucket memory and the look-aside buffer) are realized by the structure memory controller (SMC), the bucket memory system (BMS) and the look-aside buffer memory (LABM). The SMC is responsible for interfacing with the DBCCP, the KXU, and the SMIP (see Fig. 1), translating DBCCP commands, and transforming logical bucket names into physical bucket names.

We begin this section by proposing an efficient directory entry storage format based on the discussions in the previous section. We then discuss three design alternatives for the BMS. Each of the three design alternatives is based on a different technology. The first design utilizes magnetic bubble memory systems; the second is based on charge-coupled devices; and the third uses election beam addressable memories. All three technologies are emerging technologies and therein lies the rationale for presenting three design alternatives. It is not clear, at present, which of the three will ultimately develop into a commercial product. One thing, however, is clear: it will be difficult if not impossible to implement a reasonably powerful SM at a reasonable cost without the emerging technologies. The design alternatives are followed by a discussion on the BMS logic. The BMS utilizes an array of processing elements to search and manipulate the contents of a bucket. Finally, we deal with the implementation of the SMC and the LABM.

3.2.1 Data Structures for the Bucket Memory System (BMS)

In earlier sections, we noted that directory entries of keywords which are stored in a particular logical bucket (and, hence, in a single physical bucket) have identical logical bucket names. Since the logical bucket name occupies the 24 high order bits in a transformed keyword, it follows that all directory entries in a bucket would have the same 24 high order bits in their respective keyword field. Therefore, only the 24 low order bits of T(K) would actually be stored in the storage system. This is shown in Figure

10. [Recall from the discussion on KXU, that the 24 low order bits are produced by user supplied hash algorithms, and hence do not play a role in locating the directory entry of the keyword].

The index term's three components have the following lengths. Eight bits represent an MAU identifier, while two ten-bit fields are used to identify the cluster and the security atom. This makes for a total of 28 bits. The rationale for this choice is as follows. Consider the space requirements of a file. The minimum unit of allocation of mass memory is an MAU. Allocation of an MAU means an allocation of about 4 million bits of MM. A maximum allocation of 256 MAUs for a file provides a total capacity of over a billion bits which we feel is adequate for most large files encountered in practice. Thus, eight bits in the index term can uniquely identify any of the 256 possible MAUs allocated to a file. As we shall see later, the IXU maintains a translation table for each file by which the actual MAU address may be determined. Also, we do not anticipate files with more than 1,000 clusters and 1,000 security atoms. Thus the 10-bit fields are seen to be adequate to represent the anticipated cluster and security atom population in a file.

In the case of clustering/security keywords, the logic of some of the DBCCP algorithms requires that the number (or count) of clustering/security keywords that make up a cluster/security atom be made available in the index term. Thus, each index term of a clustering/security keyword has a count field associated with it. In Fig. 11, we have the formats of the various types of directory entries.

We now turn to the consideration of sizes of typical directory entries with a view to establishing certain minimum-access-unit sizes within the bucket memory. Such minimum-access units will be called modules. The size of the directory entry of a simple keyword shown in Figure 11b varies with the number of index terms in the entry. Let this number be m for simple keyword K. Then the number of bits b(K) required for storing the directory entry of a keyword K is

$$\begin{aligned} &b(K) && = |T(K)| + 16 + m \cdot |i| \\ &\text{where } |T(K)| && = 24 \text{ (low order bits of } T(K)) \\ &\text{and } |i| && = 28 \text{ (size of index term)} \\ &\text{thus, } b(K) && = 40 + 28m. \end{aligned}$$

The size of a security or clustering keyword directory entry will be larger than what we have indicated for a simple keyword directory entry.

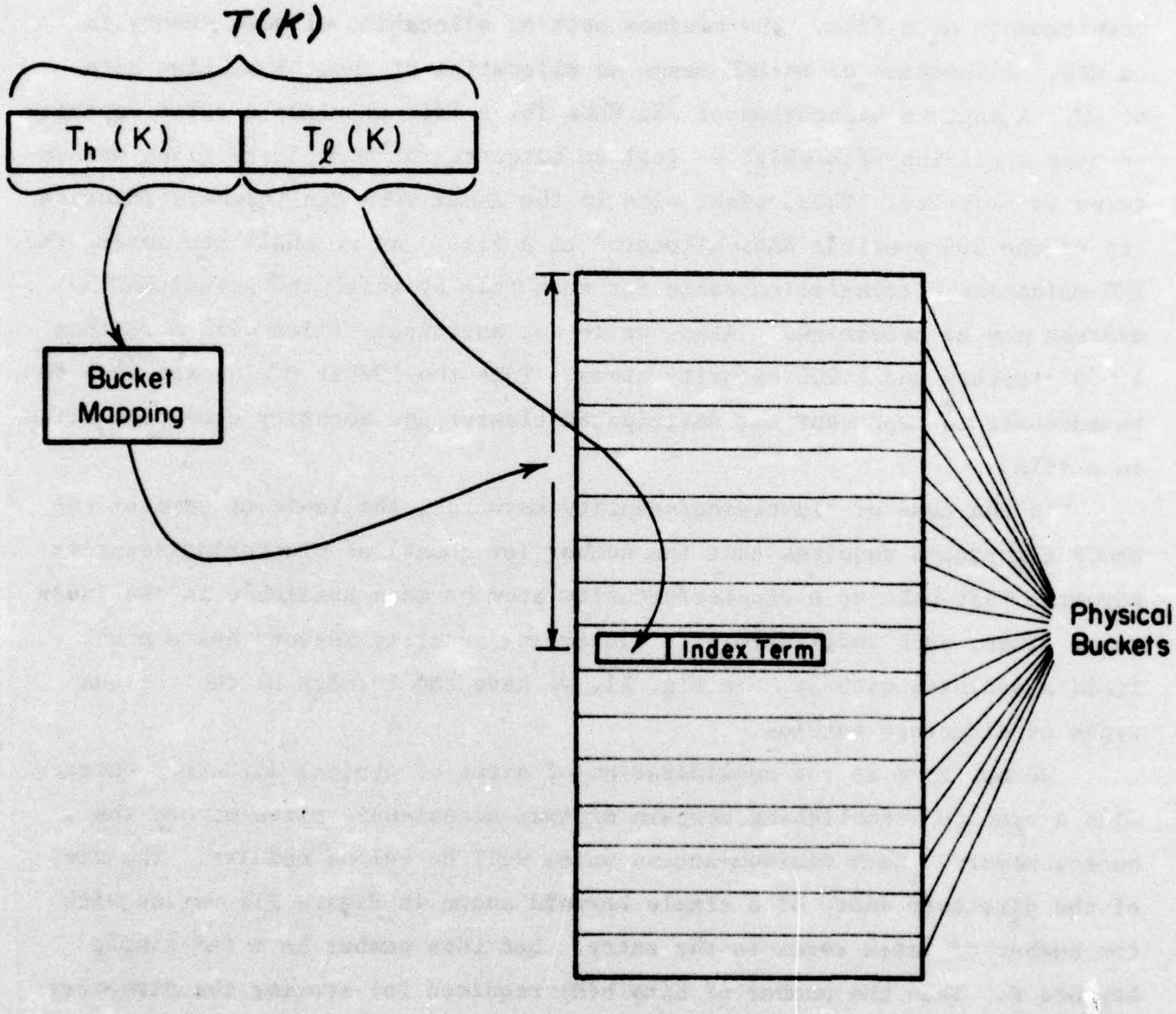


Figure 10. Storage of T(K)

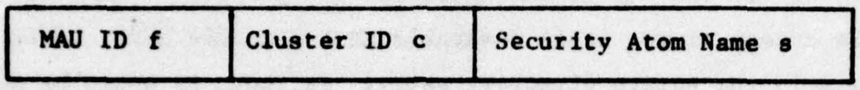


Figure 11a. Format of an Index Term

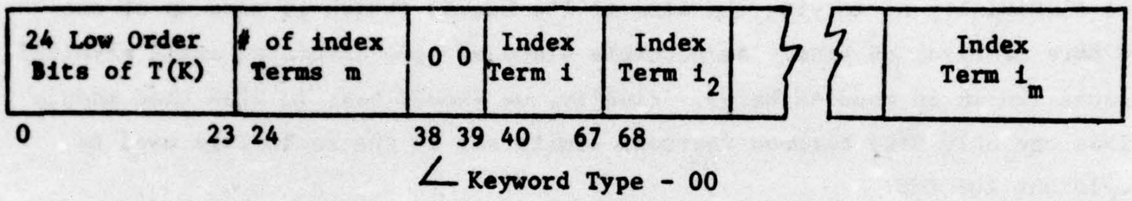


Figure 11b. Format of a Simple Keyword Directory Entry

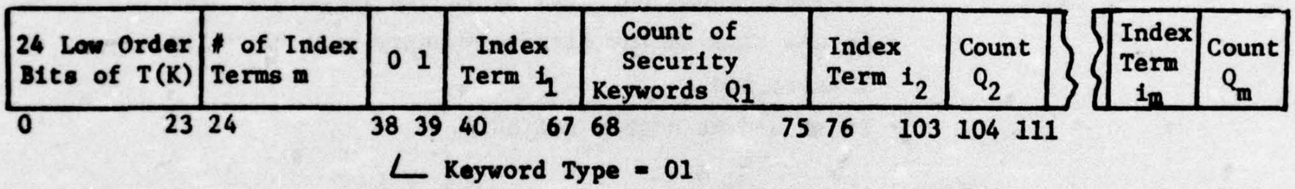


Figure 11c. Format of a Security Keyword Directory Entry

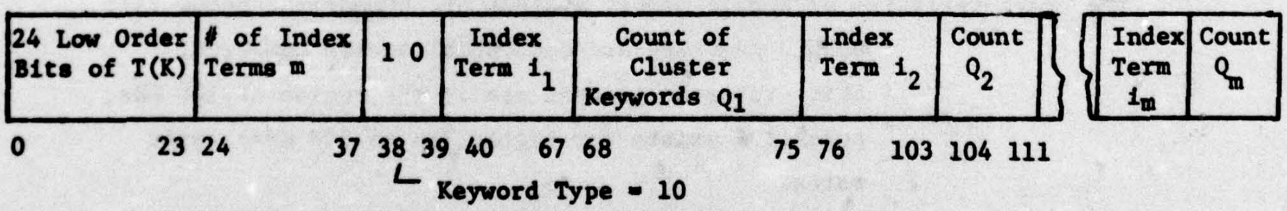


Figure 11d. Format of a Clustering Keyword Directory Entry

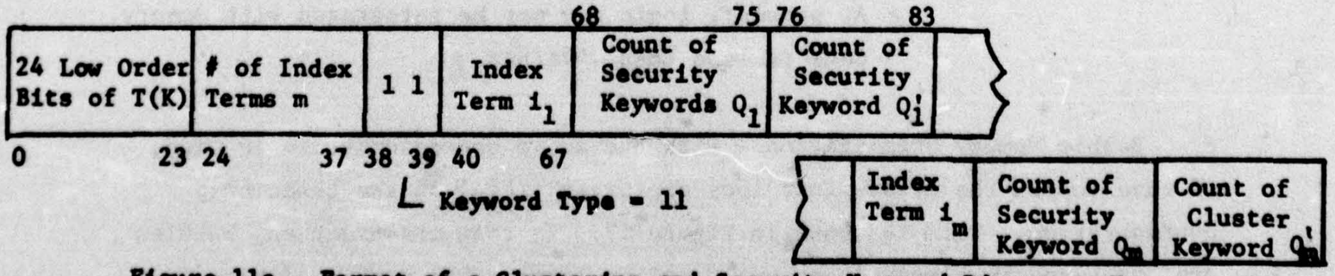

68
75 76
83
24 Low Order Bits of T(K)
# of Index Terms m
1 1
Index Term i_1
Count of Security Keywords Q_1
Count of Security Keyword Q_1'
0
23 24
37 38 39 40
67
Keyword Type = 11
Index Term i_m
Count of Security Keyword Q_m
Count of Cluster Keyword Q_m'

Figure 11e. Format of a Clustering and Security Keyword Directory Entry

But since the number of security/clustering keywords is small compared to simple ones, we can use the above figure for our purpose. In order to minimize the access times, it is desirable that a module have sufficient capacity to hold the entire directory entry. Further, it would be desirable to pack several directory entries in a single module since the number of modules to be accessed for directory entries of keywords satisfying a predicate would be minimized. On the other hand, we would like to retain the flexibility of varying the size of the bucket (which is made up of one or more modules) as finely as possible since we have advocated small physical blocks (known as modules here). Finally, we should bear in mind that module sizes can only vary between (narrow) limits set by the technology used to implement the BMS.

We are now in a position to state the requirements of the BMS. These are as follows:

- Total capacity is of 10^7-10^9 bytes,
- Typical module capacity is in the range 1-8 Kbits,
- Access time to any directory entry must be under 1 msec; and
- It should be highly reliable.

3.2.2 Technology Based Design Alternatives for the BMS

A. Magnetic-Bubble-Memory-Based Bucket Memory

The characteristics of bubble memory systems are summarized below [4]:

- Bubble memories are sequential access memories.
- Shift rates at present are in the region of 100 KHz; potential exists for higher (up to 500 KHz) shift rates.
- Bidirectional shifting is possible; bubbles may be stopped and started without incurring time penalties.
- At present, logic may not be integrated with memory.
- Cost is less than .02¢/bit.

a. Bubble Memory Organization - With the above characteristics in mind, we have chosen the major-minor loop organization [4] as the basic chip configuration. This is shown in Figure 12. In this organization, bubbles are organized in closed loops called minor loops. The number of bits W

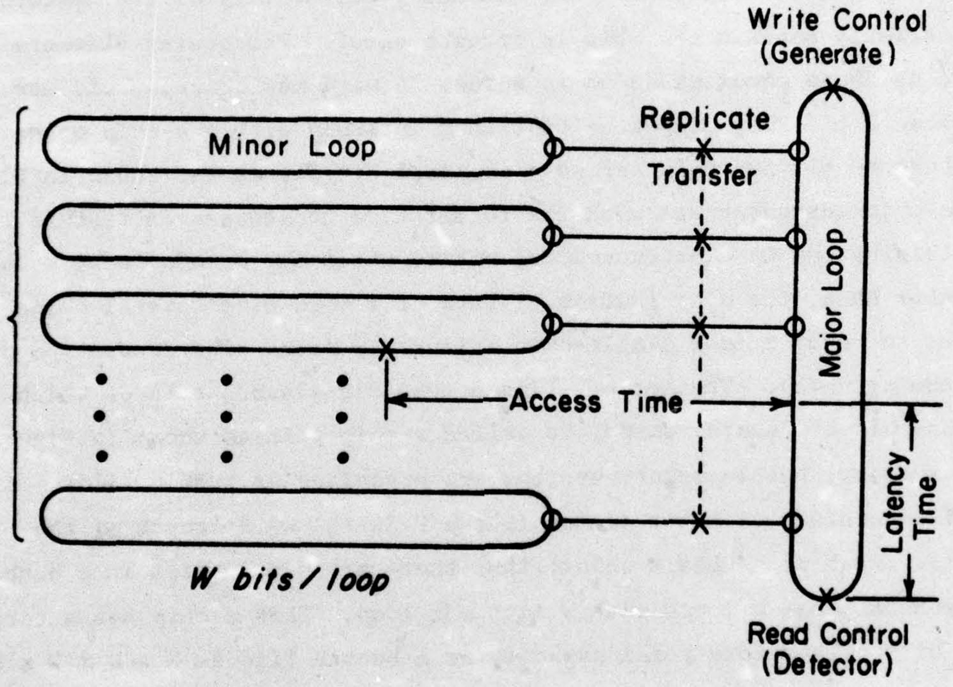


Figure 12. Major-Minor Loop Organization

per minor loop and the number of such loops N in a chip are design parameters which we shall compute later in this section. Bubbles in the N minor loops are always moved in synchronization. The presence of a bubble in a bit position constitutes a '1' and the absence of the bubble constitutes '0'. Bubble chips with the above configuration are organized into 'bit' planes. Each bit-plane consists of a set of M chips mounted on a board. A set of bit-planes, called a bubble file, is shown in Figure 13.

The concept of a bit-plane is borrowed from semiconductor memory technology and is primarily intended for enhancing reliability of the system. We shall briefly explain how this is brought about. Processing elements which access these memories do so in words. A word may comprise of, say, P bits. These P bits may either be contained entirely within a chip or be distributed one bit per chip across P separate bit-planes (as shown in Figure 13). The major disadvantage with the former type of storage is that if the chip containing the word malfunctions, entire words may become inaccessible. On the other hand, the distribution of bits of a word over several chips enables us to recover from single-chip failure by means of redundancy coding (e.g., Hamming code). The set of chips across bit-planes, each of which carries one bit of several words, is called a chip pile as shown in Figure 14.

In summary, bubble memory systems are organized as bubble files. A bubble file consists of P bit planes (where P is the word length of the processor). Each plane has M chips; thus there are M x P chips in a bubble file. Each chip has N loops with W bits per loop. Thus a chip has a total of N x W bits. Therefore total capacity of a bubble file is M x N x W x P bits.

b. Access Considerations - As seen from Figure 12, there are two time periods involved in reading out a single bit from any of the minor loops. First the bit (or bubble) has to be moved to the major loop (from its position in the minor loop). We shall call this time period the access time. Second, the bit has to be moved along the major loop till it reaches the read station. This time period is called the latency time. In the worst case,

$$t = \text{Time to read a single bit} = \underset{\substack{\text{Access}\\ \text{Time}}}{\frac{W}{2f}} + \underset{\substack{\text{Latency}\\ \text{Time}}}{\frac{N}{f}}$$

where f is the shift frequency. At the present state of technology, the shift frequency is rather low (about 100 KHz) and thus the total time to read a bit could be several milliseconds for any reasonable value of W and N.

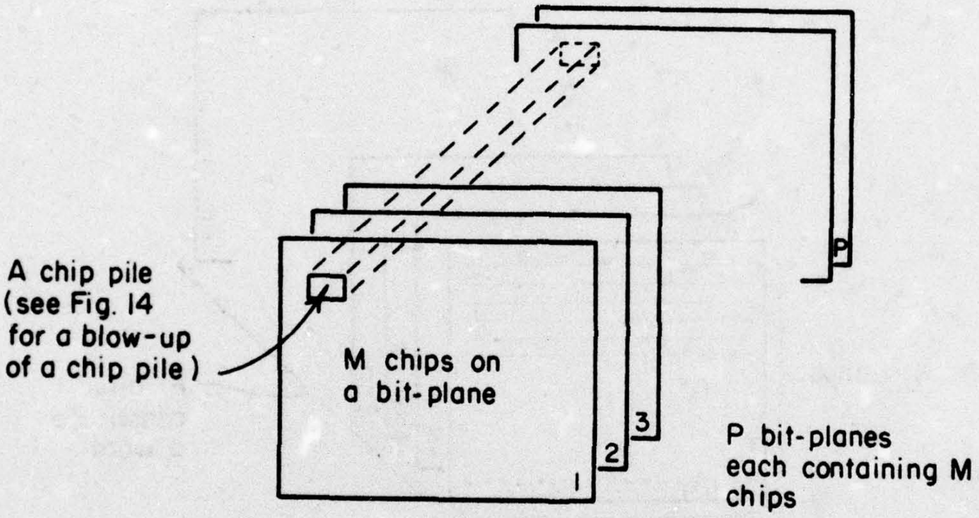


Figure 13. A Bubble File

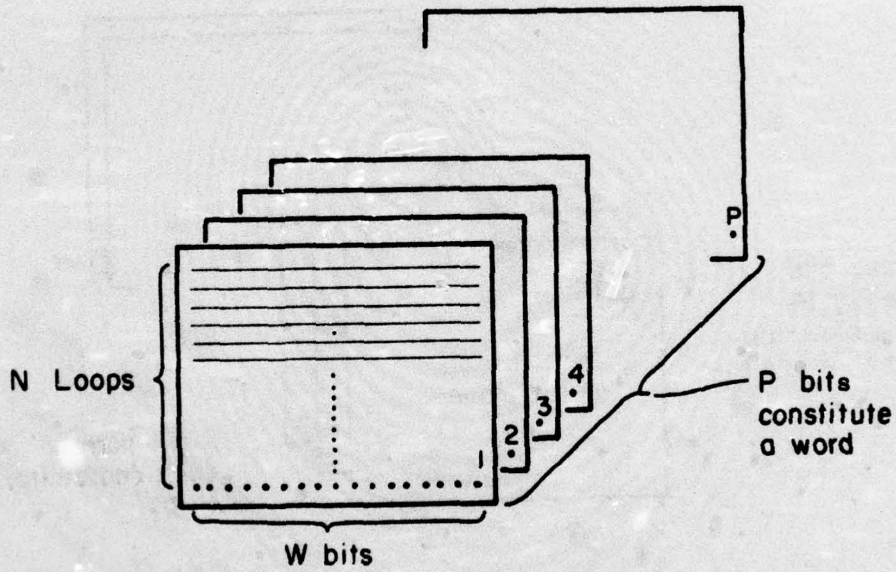


Figure 14. A Bubble Chip Pile Containing P Chips Each Containing One Bit of N × W Words

This rules out the possibility of serial readout of minor loops. Clearly, a high degree of bit-interleaving is indicated. Since a chip associated with a chip pile can be activated simultaneously, we obtain a P-fold increase in data transfer. Logically, we find that such a scheme will provide us with one word instead of one bit per readout. To further increase the transfer rate, we can activate all the M chips on a bit plane. This scheme gives us M words (of P bits each) per readout.

We are now in a position to define a module which is the smallest unit of access and determine the time required to access it.

Definition - A <u>module</u> is the set of words which can be retrieved by a complete readout of the major loops of all the chips in a bubble file. Since the contents of a major loop within a chip constitute one bit position of all the minor loops in the chip, we conclude that the number of modules in a bubble file is equal to the number of bit positions in the minor loop. The average access time to a module is (W/2)/2f. (Here we take advantage of bidirectional shifting; thus, the longest time for moving a bit from the minor loop into the major loop is W/2f. On an average this time will be one half the longest time). The time to read a module is N/f. (N is the number of minor loops in the chip. By definition a module has one bit in each and every minor loop in the chip). Thus, the average time T_M to access and read a module = $(\frac{W}{4} + N) \times f$. In our design we shall assume f to be 100 KHz. Then,

$$T_M = \frac{1}{10^5}\left(\frac{W}{4} + N\right).$$

Since a keyword directory entry is stored in a module, and since we require that a keyword entrv be retrieved within 1 ms, we have the inequality

$$T_M \leq 10^{-3}$$

or
$$\frac{1}{10^5} \times \left(N + \frac{W}{4}\right) \leq \frac{1}{10^3}$$

or
$$\boxed{\left(N + \frac{W}{4}\right) \leq 100}$$

The above inequality holds only if we assume that a keyword directory entry is found in the first module that is accessed. Since more than one module may be allocated to a bucket, it is conceivable that several modules may have to be accessed before the directory entry is located. However, if the number of processing elements is sufficiently large, then the algorithms presented in a later section will ensure that the modules allocated to a bucket are uniformly distributed among the memory units of several processing elements. This observation implies that it will be more often the case that a processing element will have to access just one module in its memory unit to locate a keyword directory entry. Under this condition, the inequality is useful for calculating values of N and W. In Table III, we have tabulated typical values of N and W satisfying the inequality in the box.

The number of processors needed for the bucket memory depends on the total capacity of the bucket memory, the module access time and processor capability. Since the capability of a processor is not an easily determinable quantity, we merely tabulate in Table IV the memory unit capacity as a function of total bucket memory capacity and number of processors. In Table V, we tabulate the number of chips required in the memory unit of a processor as a function of the chip size (from Table III) and memory unit capacity (from Table IV). A knowledge of the number of chips per memory unit will enable us to determine the number of bubble files needed and the number of chips in each of the bubble files. Let P be the size of the processor word and M be the number of chips on a bit-plane. Then, the number of bubble files is (total number of chips in memory unit) / (M x P). P is fixed when the processor is chosen. The number M is decided by trial and error within limits set by packaging technology. For example, a typical range of M would be 10-30 chips on a single board. In this design, the average retrieval time of a module is independent of the number P of bit planes or the number M of chips on each bit plane. The size of a module, however, is dependent on these parameters and is M x N x P bits. The number of modules in a bubble file is W, the number of bit positions (or bubbles) in a minor loop.

c. Design Algorithm - This design gives a step-by-step procedure to determine the various parameters discussed this far.

Input Arguments:
1. Size of SM required
2. Number of processing elements
3. Word size P of processor
4. Size of module desired within limits.

Step 1: From the size of SM and number of processor, determine the size of the memory unit attached to each of the processors by using Table IV. Call this S_M.

Step 2: From the size of the module desired and the word size P of the processor, determine the product M x N where M is the number of chips on a bit plane and N is the number of minor loops in the chip, since the module size is M x N x P.

Table III. Typical values of N and W which satisfy access constraints.

W	N	W x N = chip size
32	92	2944
64	84	5376
96	76	7296
128	68	8704
160	60	9600
192	52	9984
224	44	9856
256	36	9216
288	28	8064
320	20	6400
352	12	4224

Table IV. Memory capacity of a processor (in K bytes) as a function total SM capacity and number of processors.

Number of processors	SM Capacity (bytes)		
	10^7	10^8	10^9
32	320	3200	32000
64	160	1600	16000
128	80	800	8000
256	40	400	4000

Table V. Number of chips as a function of chip size and memory unit capacity (K bytes).

Memory Unit Capacity in Kbytes	Chip size (bits)											
	2944	5376	7296	8704	9600	9984	9856	9216	8064	6400	4224	1536
40	109	60	44	37	34	33	33	35	40	50	76	209
80	218	120	88	79	68	66	66	70	80	100	152	418
160	436	240	176	148	136	132	132	140	160	200	304	836
320	872	480	352	296	272	264	264	280	320	400	608	1672
400	1090	600	440	370	340	330	330	350	400	500	760	2090
800	2180	1200	880	740	680	660	660	700	800	1000	1520	4180
1600	4360	2400	1760	1480	1360	1320	1320	1400	1600	2000	3040	8360
3200	8720	4800	3520	2960	2720	2640	2640	2700	3200	4000	6080	16720
4000	10900	6000	4400	3700	3400	3300	3300	3500	4000	5000	7600	20900
8000	21800	12000	8800	7400	6800	6600	6600	7000	8000	10000	15200	41800
16,000	45600	24000	17600	14800	136000	13200	13200	14000	16000	20000	30400	83600
32,000	87200	87200	35200	29600	27200	26400	26400	27000	32000	40000	60800	167200

Step 3: Using S_M and choosing available chip size, determine the number of chips required from Table V. Call it C_M.

Step 4: For the chip size used in Step 3, look up Table III to determine the values of N and W. If the chip size is not found in Table III, then go back to Step 3 and choose another chip size.

Step 5: Determine M where M = Module Size / (P x N).

Step 6: Determine number of chips on a bubble file BF_s when BF_s = M x P.

Step 7: Compute the number of bubble files as $\lceil C_M / BF_s \rceil$.

d. Design Example - We now give an example using the above design algorithm.

Input Arguments: 1. Structure of Memory Size = 10^8 bytes

2. Number of Processors = 128

3. Word Size = 16 bits

4. Module Size = 2 Kbytes.

Step 1: From Table IV we find S_M = Size of memory unit = 800 Kbytes.

Step 2: Module size = 2 Kbytes = 2 x 8 x 1024 bits = M x N x P. Therefore,

$$M \times N = \frac{2 \times 8 \times 1024}{16} = 1024.$$

Step 3: Choose chip size = 9856. Then, from Table V, we have C_M = 660.

Step 4: For the chip size = 9856, we have from Table III N = 36, W = 224.

Step 5: $M = \frac{\text{Module Size}}{P \times N} = \frac{2 \times 8 \times 1024}{16 \times 36} = \frac{1024}{36} = 29.$

Step 6: BF_s = Number of chips per bubble file = 29 x 16 = 464.

Step 7: Number of bubble files $= \lceil C_M / BF_s \rceil = \lceil 660 / 464 \rceil = 2.$

Summary of design example

Number of bubble files = 2.

Number of chips on each bubble file = 464.

Capacity of bubble file = 9856 x 464 bits = 571648 bytes = 570 Kbytes.

Capacity per memory unit = 1140 Kbytes.

Chip details

N = number of minor loops = 36.

W = number of bits per loop = 234.

Module Size = 29 x 36 x 16 = 16864 bits = 2108 bytes.

Number of modules per bubble file = 224.

Number of modules per memory unit = 448.

Number of modules in the SM = 448 x 128 = 57344.

e. Reliability consideration - The organization outlined in the above subsection is well suited for incorporating error detecting and correcting codes. The incorporation of one such code will be briefly described here. In Table VI, we tabulate the number of error code bits required for various word sizes. For any word size p, we add k (error code) bit planes to the p (data storing) bit planes. Input and output error code generators are then added to identify the bit plane in which a fault might have occurred. On identification, the operator is alerted to replace the faulty bit plane with a fault-free plane and reload the new bit plane with the helf of the error code generators. We observe, that the detection, correction and replacement of malfunctioning components are achieved without having to place the system off-line.

The need for such reliability measures at a cost about 37.5% of a memory system with 16-bit word may be justified in terms of the MTBF (mean time between failures) of the system where

$$\mathrm{MTBF} = \frac{1}{dx}$$ where d is the number of chips and x is

the failure rate. In the design example, we had a chip count of about 1000 per memory unit. Since we had 128 memory units corresponding to 128 processing elements, the total chip count is about 128000. If we assume a chip failure rate of 0.1% per 1000 hours, then we have an MTBF of about 8 hours. With such an MTBF, the reliability provision looks quite desirable.

B. Charge-Coupled-Device-Based Bucket Memory

Let us enumerate the main characteristics of charge-coupled devices (CCDs).

- Charge-coupled devices, like bubble memories, are sequential access memory devices.
- Shift rates vary from at least 2 MHz to 10 MHz.
- Memory needs to be refreshed periodically.
- On-chip logic is possible.
- High stand-by power is required.
- Cost per bit is between 0.05 to 0.02¢.

The technology of CCDs is in fact the technology of semiconductor devices, and many of the innovations and experiences in LSI fabrication are applicable to charge-coupled devices. In addition, because the structure of CCDs is simpler than that of other LSI products [6], one should expect higher chip

Table VI. Single Error Correction and Double Detection.[12]

Error Code Bits	k	5	6	7	8
Data word size	p	4-10	11-25	26-56	57-119
Total Bits	n = p + k	9-15	17-31	33-63	65-127
Redundancy Ratio	n/p	2.25 - 1.5	1.55 - 1.24	1.27 - 1.13	1.14 - 1.07

(minimum Hamming Code distance: 4)

density. Currently announced chips have a capacity of 16K bits, while a 65 K-bit prototype has been built [11]. It is estimated that the 128 K bit chip is a distinct possibility with improved fabrication techniques [6]. With this information in the background, we begin our design by tabulating in Table VII the total number of chips required in the bucket memory as a function of chip size and total bucket memory size. We note that the bucket memory is actually divided equally among n processors. In Table IV, we have tabulated the memory unit size as a function of the number of processors and total memory size. For every memory unit size tabulated therein, we tabulate in Table VIII the number of chips needed as a function of the chip size.

A number of chip designs have been suggested [1,11]. In choosing a design to suit our needs, we must carefully evaluate the tradeoffs involved in each of the designs. For large bucket memories, say 10^9 bytes, we must use large size chips ($\geq$ 32K) if we wish to keep the number of chips down to reasonable levels. This is important from a reliability standpoint, since the higher the chip count, the lower the MTBF for a given failure rate. Large size chips imply high densities. Designs which feature large densities do so by reducing the frequency of operation in order to keep the power dissipation of the chip low and by reducing the number of refresh stages, I/O stations and other circuitry which take up space in the chip at the expense of storage elements. A typical case is the SPS (serial-parallel-serial) organization which can support the highest densities in a chip. However, such high densities are achieved at the price of an increase in the access time to a bit within the chip. This is true of the SPS organization. For example, for a chip size of 64 Kbits and a shift rate of 10MHz the average access time is 3.2 msec. Since the SPS organization does not support multiple loops which can be individually accessed, it is not possible to shorten the access times by clever organization of modules. We, therefore, conclude that the price to be paid for realizing large buckets in terms of higher access times is not acceptable.

For small and medium size bucket memories we can use chips of sizes 8 - 16 Kbits, which imply that packing densities can be considerably lower. In Figure 15, we present the so-called LARAM (Line Addressable RAM) organization [1] for implementing small and medium size bucket memories.

a. LARAM organization - This organization has the following properties:

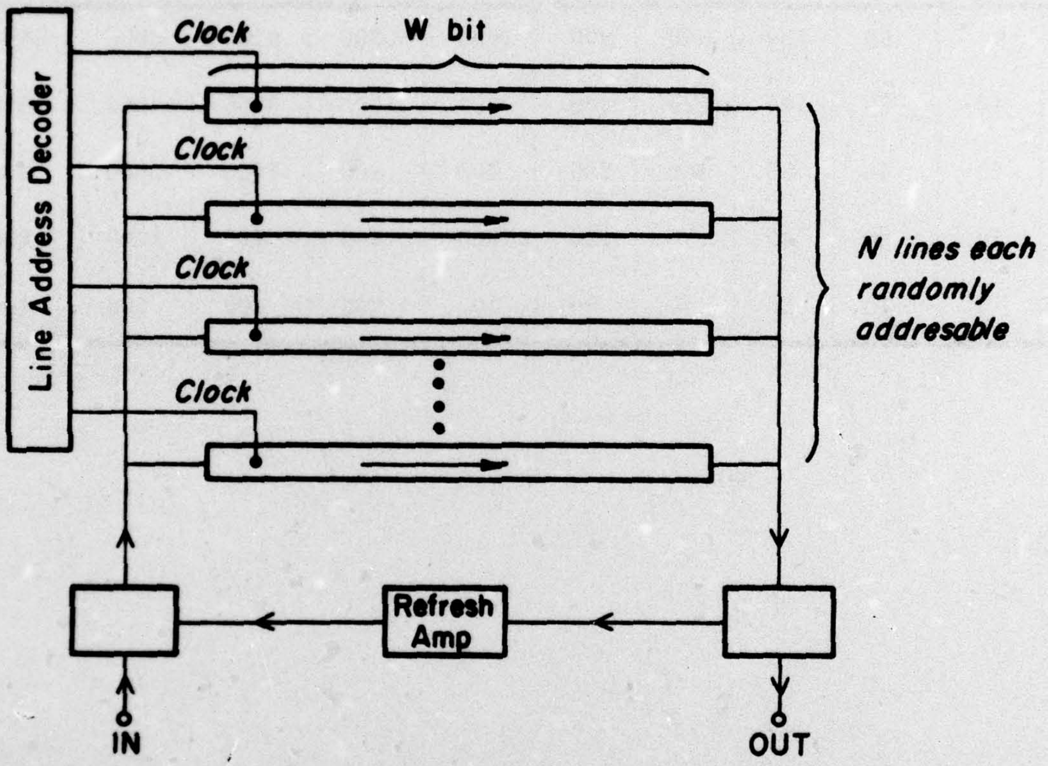
1. Random access to any CCD line is possible.
2. Access time to a line is negligible compared to read-out time of the line.
3. On-chip line address decoder is available.

Table VII. Number of chips as a function of chip size and total bucket memory size (bytes).

Chip Size (bits)	SM Capacity		
	10^7	10^8	10^9
4K	20,000	200,000	2,000,000
8K	10,000	100,000	1,000,000
16K	5,000	50,000	500,000
32K	2,500	25,000	250,000
64K	1,250	12,500	125,000

Table VIII. Number of chips as a function of memory unit size and chip size.

Chip Size (bits)	Memory Unit (Kbyte)										
	40	80	160	320	400	800	1600	3200	4000	8000	16000
4K	80	160	320	640	800	1600	3200	6400	8000	16000	32000
8K	40	80	160	320	400	800	1600	3200	4000	8000	16000
16K	20	40	80	160	200	400	800	1600	2000	4000	8000
32K	10	20	40	80	100	200	400	800	1000	2000	4000
64K	5	10	20	40	50	100	200	400	500	1000	2000



Lines are moved either for refresh or access purposes only.

Figure 15. LARAM Organization

4. High bit rates and short access times are possible.

5. Fabrication is complicated resulting in low density.

Chip sizes are assumed to be in the range 8-16K bits. As in the case of bubble memory design, we organize chips into CCD files. Each file is made up of P (data) bit planes and k (error code) bit planes. (Here again, P is the width of a processor word and k is determined as in Table VI). Each bit plane is populated by M chips where M is one of the design parameters. Unlike bubble memory loops, however, LARAM lines are not synchronized, and, thus, the concept of modules corresponding to bit positions (as in the case of the bubble memory system) does not hold in LARAM design. Instead, the contents of an entire line will form part of a module. Assuming a 2 MHz basic clock rate, the time required to read out an entire line of various sizes is given in Table IX. As can be observed from the Table IX, even for a large loop (with 512 bits) the readout time is small compared to bubble memory readout times. Let the line size be W bits. A module is defined to be the set of bits contained in P lines each of which is on a bit plane. The module size is, therefore, P x W. If there are N lines in a chip giving a chip capacity of N x W, then each chip in a chip pile would hold one bit of all the words constituting N modules. A chip pile is defined in exactly the same manner as in the bubble memory system. The major difference between the CCD chip pile and the bubble chip pile is the manner in which the modules are defined. In the bubble memory design, there were W modules distributed in M chip piles. In the LARAM-based CCD system there are N complete modules per chip pile. Each module has W words and each of the W words is distributed across P bit planes. We shall call the set of modules defined by M CCD chip piles, a CCD file as depicted in Fig. 16. As in the case of the bubble memory system, we need to determine the values of N and W. In Table X, we tabulate the number W of bits per line as a function of the number of lines and chip size.

A number of observations can be made at this point as a prelude to the design algorithm. First, the level of interleaving is limited to that required to form a single word. Recall that in the case of the bubble design we had a much higher level of interleaving because of the slower shift rate of the bubbles. Second, each individual line in a chip can be addressed to the exclusion of other lines by means of on-chip address decoding circuitry. Again, this is in contrast to the bubble design where minor loops could not be addressed conceptually in three steps. In step one, we select the CCD file. Then, the chip within the CCD file is selected. In step three, the line

Table IX. Line size and read-out times (2 $\mathrm{MH_Z}$ clock).

Line Size	Read-out Time (μsec)
64	32
128	64
192	96
256	128
320	160
384	192
448	224
512	256

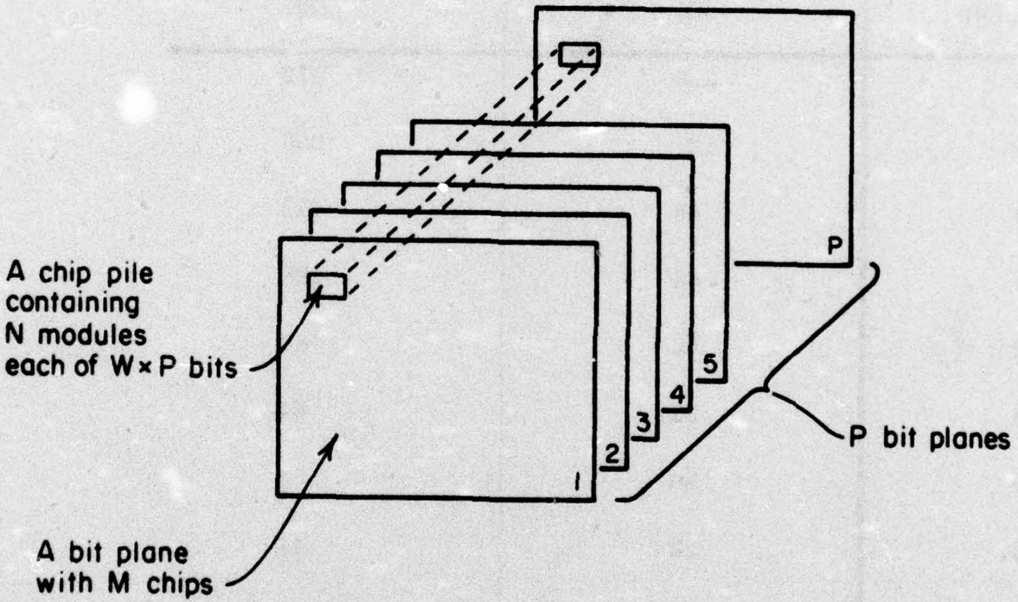


Figure 16. A CCD File

Table X. Number of bits per line (W) as a function of size and number of loops (N).

Number of loops (N)	Chip size	
	8K	16K
32	256	512
64	128	256
96	86	172
128	64	128
192	43	86
256	32	64
320	26	53
384	22	44
512	16	32

constituting a part of the module is selected. In implementation, the module address is broken into three components - one for selecting the CCD file, one for selecting the chip within the CCD, and one for selecting the line within the chip. Fourth, since the data is transferred quite rapidly (a word every 0.5 μsec.), it may not be possible for the processor to process the information on-the-fly. It may then become necessary to buffer the data. Fifth, as a result of random access to a line, access time to a module is negligible compared to the access time in bubble system.

b. Design Algorithm

Input Arguments:
1. Bucket Memory Size
2. Module Size
3. Word size of processor
4. Number of processors

Step 1: From bucket memory size and number of processor, determine size of memory unit attached to each of the processors by using Table IV.

Step 2: Choose a chip size - 8K or 16K. Calculate total number of chips by using Table VII.

Step 3: Use the word size and module size to determine the number of bits per line with the help of the following equation:

Module Size = Number of bits per word x number of bits per line

Step 4: Use the number of bits per line determined in Step 3 and the chip size determined in Step 2 to determine the number of loops per chip from Table X.

Step 5: Determine read-out time for a module from Table IX.

Step 6: Knowing the chip size (Step 2) and size of memory attached to a processor, determine the number of chips in the memory unit by using Table VIII. This is distributed in one or more CCD files. The number of CCD files can be determined by iterative calculation as follows:

Let number of CCD files by 1, then,

$$\text{Number of chips per bit plane} = \frac{\text{Number of chips in a memory unit}}{\text{Word size x No. of CCD files}}$$

If the number of chips per bit plane is in the range 10-30, go to Step 7. Otherwise, increase the number of CCD files by one and recalculate.

Step 7: Number of modules = (Number of CCD files) x
(Number of chips on a bit plane) x
(Number of loops per chip).

c. Design Example

Input: Total bucket memory size = 10^8 bytes
Module size desired = 8 Kbits
Word size of processor = 16 bits
# of processors = 128

Step 1: Size of Memory Unit attached to a processor = 800 Kbytes from Table IV.

Step 2: Chip size chosen = 16 K bits.
Total number of chips = 50,000 from Table VII.

Step 3: Number of bits per loop, $W = \frac{\text{Module Size}}{\text{Word Size}} = \frac{8 \times 1024}{16} = 512.$

Step 4: From Table XI, the number of loops, N, is 32.
Step 5: From Table IX, read-out time of a module is 256μsec.

Step 6: Then, the number of chips be bit plane = $\frac{50,000}{16 \times 1 \times 128} \cong 25.$

Step 7: Number of modules per processor = 1 x 25 x 32 = 800.
Number of modules in bucket memory = 800 x 128 = 102,400.

Summary of design example

Number of CCD files = 1

Number of chips on each file = 25 x 32 = 800

Capacity of a CCD file = 25 x 16 x 16 = 6400 Kbits = 800 Kbytes

Capacity per memory unit = 800 Kbytes

Chip details

N = number of addressable loops = 32

N = number of bits per loop = 512

Module Details

Module Size = 512 x 16 bits = 8 Kbits

Number of modules per CCD file = 800

Number of modules per memory unit = 800

Number of modules in BMS = 800 x 128 = 102400

d. Reliability Considerations - Much of the discussion on reliability features for bubble memories are applicable here also. In addition, we have to consider the volatility of the CCD memories. CCD memories need refreshing periodically. In the event of a power outage, standby power must be automatically switched on to ensure retention of bucket memory contents.

C. Electron-Beam-Addressable-Memory-Based Bucket Memory

The characteristics of electron beam addressable memories (EBAM) are as follows [9]:

- Memory bits are organized as static locations on an electron beam sensitive surface.
- Memory is accessed by positioning an electron beam over the required location and scanning the area with the beam.
- Access to any location is at worst around 30μsec, if there is a change in the command mode; much less (10μsec), if there is no change in the command mode.

- Data transfer rate is around 10 MHz.
- Each "tube" of memory is conceptually equivalent to a bank of memory. Typical sizes of these "tubes" are 30 million bits.
- Cost is under 0.01¢/bit.

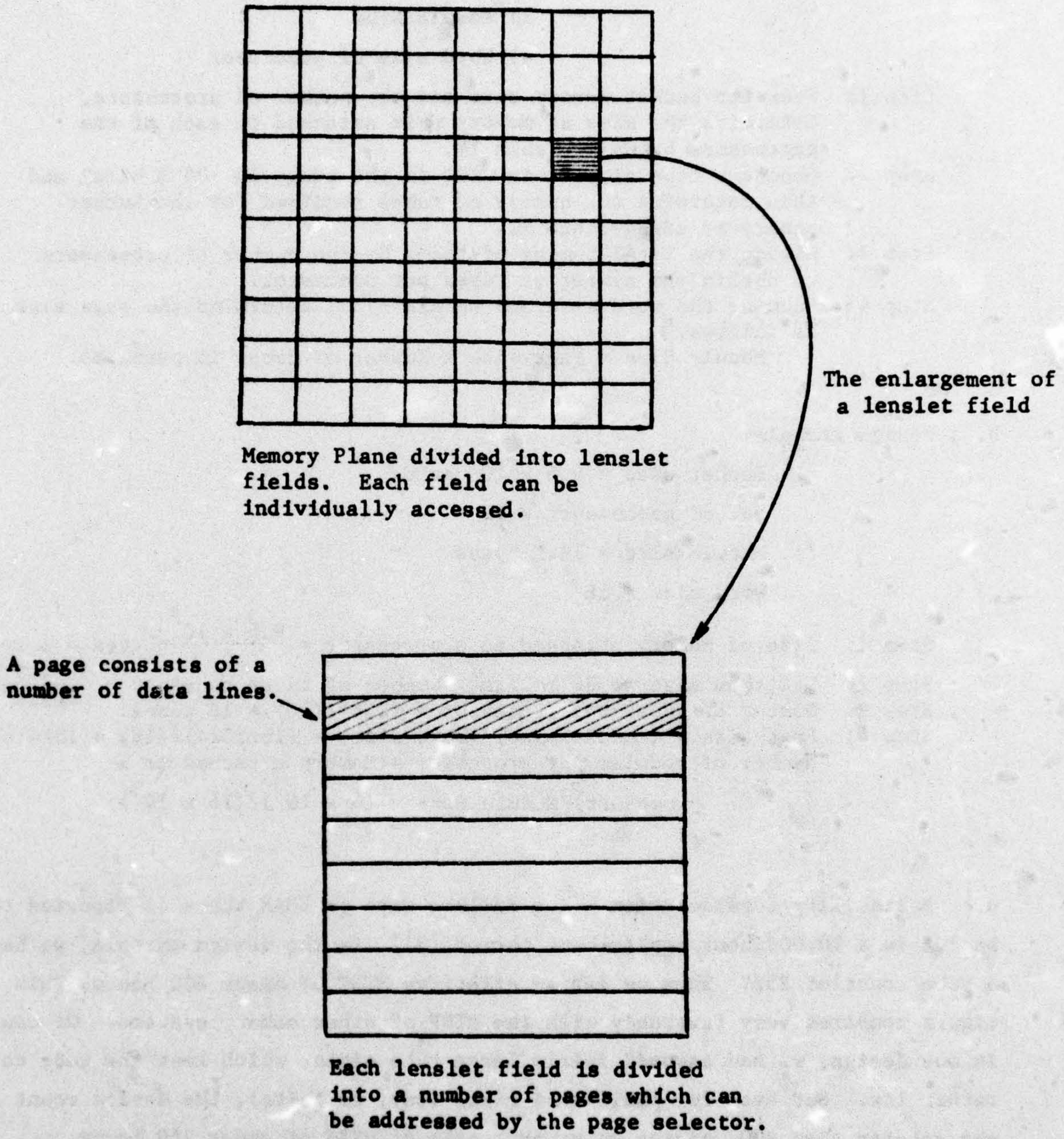
Of all the technologies considered so far, EBAMs have the best access times and seem to be truly applicable to large buket memories. As in other designs we will consider a range of tube sizes and calculate the number of tubes needed. Table XI tabulates the results of such calculations. Each word of a module is distributed over a set of tubes just as in the case of the other designs discussed thus far. The surface on which the election beam is allowed to strike is known as the memory plane. Thus a memory plane is analogous to a bit-plane in our earlier designs. Each memory plane is divided into a number of lenslet fields (see Fig. 17). A lemslet field is analogous to chips in our previous design. A lenslet field is further divided into pages. A module address therefore has two components - a field address and a page address. The size of a module can be varied within wide limits. This is because the memory plane of EBAM is unstructured (unlike the bit planes of CCDs and bubble memories) and the electron beam may be directed to any spot in the memory plane without incurring significant time penalties.

The electronics required by EBAM (i.e., the field and page select amplifies, the memory plane bias circuits and the power supplies) can be shared between tubes in a multiple tube system. Thus, the cost per bit in a multiple tube system could be substantially lower than in a single tube system. Multiple tube systems can be operated in serial as well as parallel mode. In serial mode of operation, only one tube is active at any instant of time, but in parallel mode of operation, all the tubes are active concurrently. The parallel mode of operation gives rise to a high rate of data transfer while reducing the amount of electronics that can be shared. We have chosen the parallel mode of operation not because of the resultant high data rate--the data rate of a single tube is adequate for our purposes--but because our modules are bit-interleaved to form words.

a. **Design Algorithm for EBAM-based Bucket Memory** - The design algorithm for EBAM-based bucket memories is simpler than those of bubble and CCD designs.

Table XI. Number of tubes as a function of tube size and bucket memory size.

Bucket Memory M-bits	10^7	10^8	10^9
10	8	80	800
20	4	40	400
30	3	27	267



The enlargement of a lenslet field

Memory Plane divided into lenslet fields. Each field can be individually accessed.

A page consists of a number of data lines.

Each lenslet field is divided into a number of pages which can be addressed by the page selector.

Figure 17. Memory Plane and Lenslet Fields of an EBAM

Input Arguments: 1) Bucket Memory Size
2) Number of processors
3) Module Size
4) Word size of processor

Step 1: From the bucket memory size and the number of processors, determine the size of memory unit attached to each of the processors by using Table IV.

Step 2: Choose a tube size (this will in the range 10 -30 M bits) and then determine the number of tubes required for the bucket memory by using Table XI.

Step 3: Divide the total number of tubes by the number of processors to obtain the number of tubes per processor.

Step 4: Knowing the word size and module size, determine the page size as follows:

Module Size = Page size x Number of tubes in parallel.

b. Design Example

Bucket size = 9.6×10^{8} bytes

No. of processors = 16

Module size = 16 K bytes

Word size = 16

Step 1: Size of Memory attached to a processor = $\frac{9.6 \times 10^{8}}{16}$ bytes = 6×10^{7} bytes.

Step 2: Let tube size be 30 M bits. Number of tubes required = $\frac{9.6\text{x}10^{8}\text{x}8}{30\text{x}10^{6}}$ = 256.

Step 3: Number of tubes per processor = (256)/(16) = 16 tubes.

Step 4: Page size = (Module Size)/(Word Size) = (16x1024)/(16) = 1024 bytes.
Number of modules per processor =(Memory attached to a processor)/Module Size = $(6 \times 10^{7})/(16 \times 10^{3})$
= 375

c. Reliability Consideration - The failure rate of EBAM tubes is reported to be 20% in a 20,000 hour replacement period [9]. In the design example, we had a tube count of 256. Thus we get an effective MTBF of about 400 hours. This figure compares very favorably with the MTBF of other memory systems. Of course, in our design, we had assumed fairly large tube sizes, which kept the tube count rather low. But even for small tube sizes (say, 10 Mbits), the device count is not greater than 800, giving us a worst case of MTBF of about 250 hours.

D. BMS Designs in Retrospect

The main objective of the design exercises in the preceding sections was to investigate the suitability of the three leading contenders for the realization of the bucket memory. From these exercises we have learned a number of things. Let us elaborate. All three technologies may be applicable to the problem at hand; however, the range of applicability is different for different technologies. For example, bubble memories and CCDs are feasible only for realizing small to medium size bucket memories. EBAM systems on the other hand, are economical only for larger bucket memories. In the case of bubble systems, the shift rate is too low, so we cannot have very large size loops or a great many number of loops in one chip. This implies small size chips, which in turn increases device count. We saw earlier that device count has a profound effect on reliability. In the case of CCDs, we found that increasing chip density meant sacrificing on access times--something that we can ill afford. Hence, CCDs become doubtful starters for large bucket memories. Why is EBAM economical only for large systems? Because, if the bucket memory size is small, the number of tubes per processor is proportionately small. Since the tubes attached to a processor share the electronics, it is uneconomical to have a low level of sharing. Further, if the number of tubes per processor falls below the word size, we might encounter reliability problems. The reader may observe that all the problems described above have their origin in access times and device capacity. If the access time of a memory organization using a particular technology is not within the constraints imposed by our application, we cannot use that organization. If the capacity of the smallest integral unit of memory realized by a technology is large, then we cannot use that technology economically to build small memory systems.

In our design algorithms, we have not calculated the cost of each of the systems which should be part of any viable design. Such analysis is being conducted in conjunction with an analysis of the architecture. The presentation of the above design should be regarded as a first level attempt at utilizing emerging technologies.

3.2.3 The Processing Elements of (PEs) of the BMS

A. The Number and Nature of the PE - The three design algorithms presented earlier require the number of processing elements as an input argument. This number cannot be arbitrarily chosen. For example, in case of an EBAM-based 10^8-byte bucket memory, the number of processors for a tube size of 30M bits cannot be greater than 27 (see Table XI). In other words, an upper bound on the number of processors that may be chosen is imposed by the size of memory unit that can be attached to a processor. In the case of EBAM systems, a tube is the smallest memory unit that can be attached to a processor. A lower bound is obtained by considering the processing capacity of the processor and the retrieval time of the storage system. Recall that we required a 1 ms retrieval time for an arbitrarily selected keyword directory entry. If several modules have to be searched on the average in order to retrieve a directory entry, then it may be desirable to increase the number of processor. Thus, a bucket can be worked upon by a large number of processors. Each of the processors, then, need to search over a smaller area, possibly a single module.

In choosing processors to manipulate the bucket memory, consideration should be given to the compatibility of the processor and memory. For example, the access times and transfer rate in case of EBAM systems are very good. Hence, fairly powerful and fast processing elements must be employed. On the other hand, magnetic bubble memory systems have poor access and transfer times. Therefore, slower processors may be adequate. Also, the decision of whether the data in the BMS is to be processed on-the-fly or in a buffered mode can be made only when the speed compatibility issue is determined. In the ensuing discussions, we describe the data structures maintained by each of the processors and the algorithms executed by them. These discussions are independent of processor technology (i.e., independent of whether the processing elements are made of array logic (LSI) or random logic (MSI)). The algorithms are designed to be executed on-the-fly. However, they may be executed on buffered data also if the processor is unable to keep up with the retrieval speed of the storage system.

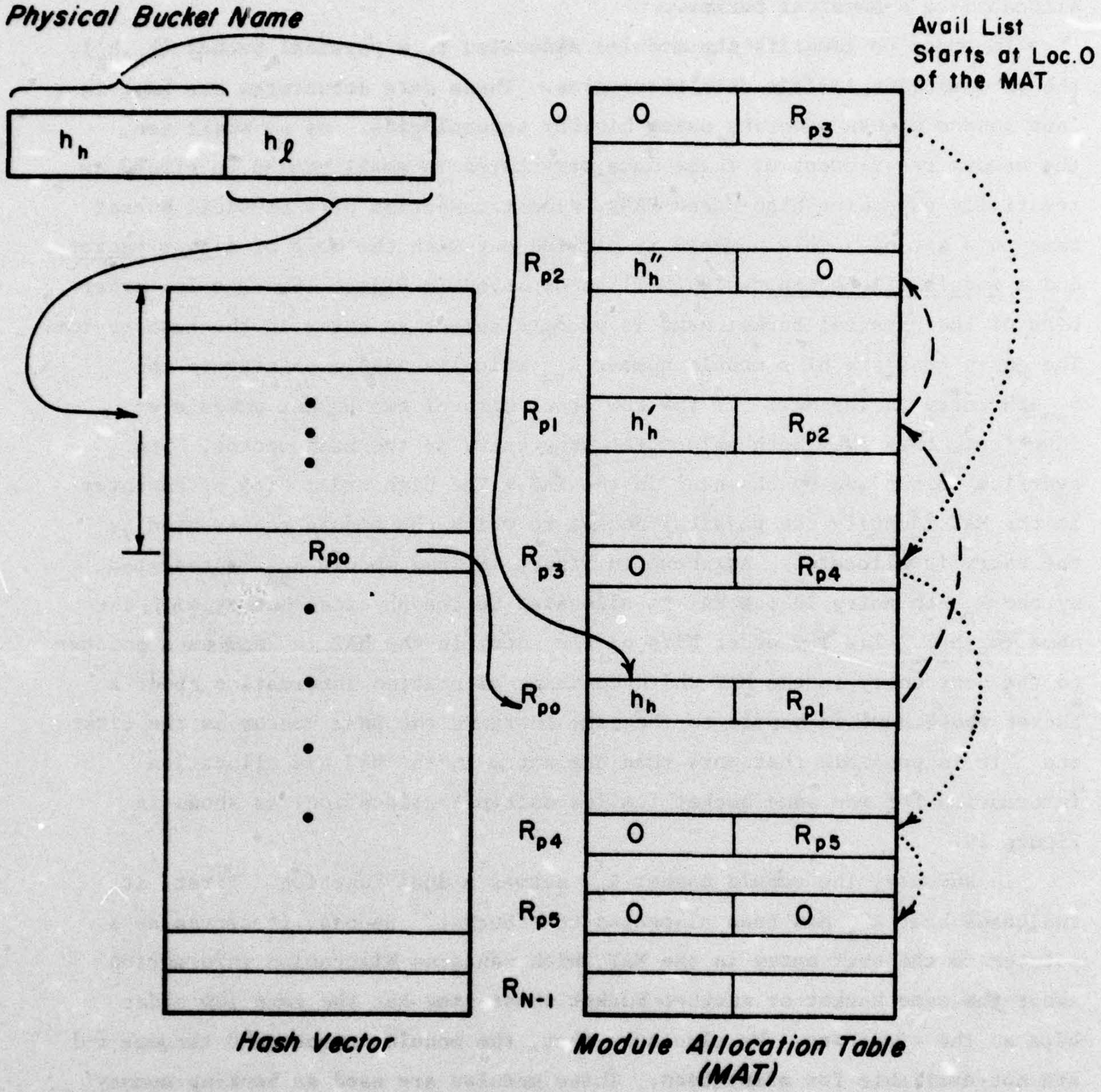
B. Data Structures Maintained by PE

Earlier, we have made two important statements regarding physical buckets. First, buckets must be of variable size which are realized with relatively small physical blocks. Second, in the design of the BMS storage, we define modules to be the smallest accessible units. Thus, in the PEs, we need a mechanism to allocate modules to physical buckets and access the modules allocated to a physical bucket.

In order to identify the modules allocated to a physical bucket (h_h, h_ℓ), the PE maintains certain data structures. These data structures are kept in fast random access memories using bipolar technologies. As we shall see, the memory requirement of these data structures is small enough to afford us relatively expensive high-speed RAMs. The translation of a physical bucket name to a set of module numbers is carried out with the help of a hash vector and a module allocation table (MAT) as depicted in Figure 18. The low order bits of the physical bucket name is used to select an entry in the hash vector. The entry consists of a module number R_{po} which is also a pointer to the R_{po}-th entry in the MAT. If the low order bits of two bucket names are identical, then they both select the same entry in the hash vector. The overflow is handled by chaining in the MAT. The high order bits of an entry in the MAT identify the physical bucket to which the module represented by the entry is allocated. As shown in Figure 18, the module R_{po} represented by the R_{po}-th entry in the MAT is allocated to the physical bucket with the name (h_h, h_ℓ). The low order bits of the entry in the MAT is used as a pointer to the next entry in the MAT which contains allocation information about a bucket whose name is mapped to the same entry in the hash vector as the first one. It is possible that more than one entry in the MAT has allocation information for the same bucket (called multiple allocation) as shown in Figure 19.

In summary, the module number R_{Pi} serves a dual function. First, it indicates that R_{Pi} has been allocated to a bucket. Second, it serves as a pointer to the next entry in the MAT which contains allocation information about the same bucket or another bucket whose name has the same low order bits as the first one. We also note that, the modules numbered 0 through r-1 are not available for allocation. These modules are used as back-up memory for the data structures of the PE. The value of r is determined by the number of modules in a memory unit and the size of each of the modules. The location 0 of the MAT is used as the head of a list of available modules.

In order to keep track of the amount of space available in each of the modules, the PE maintains a module space table (MST) as shown in Figure 20. The n-th entry in the MST records the number of bytes available in the n-th module.



Location 0 of MAT is not allocated to any bucket. It is used to claim all the available modules as shown by the dotted line.

Figure 18. Mapping a Physical Bucket Name into a Set of Module Numbers.

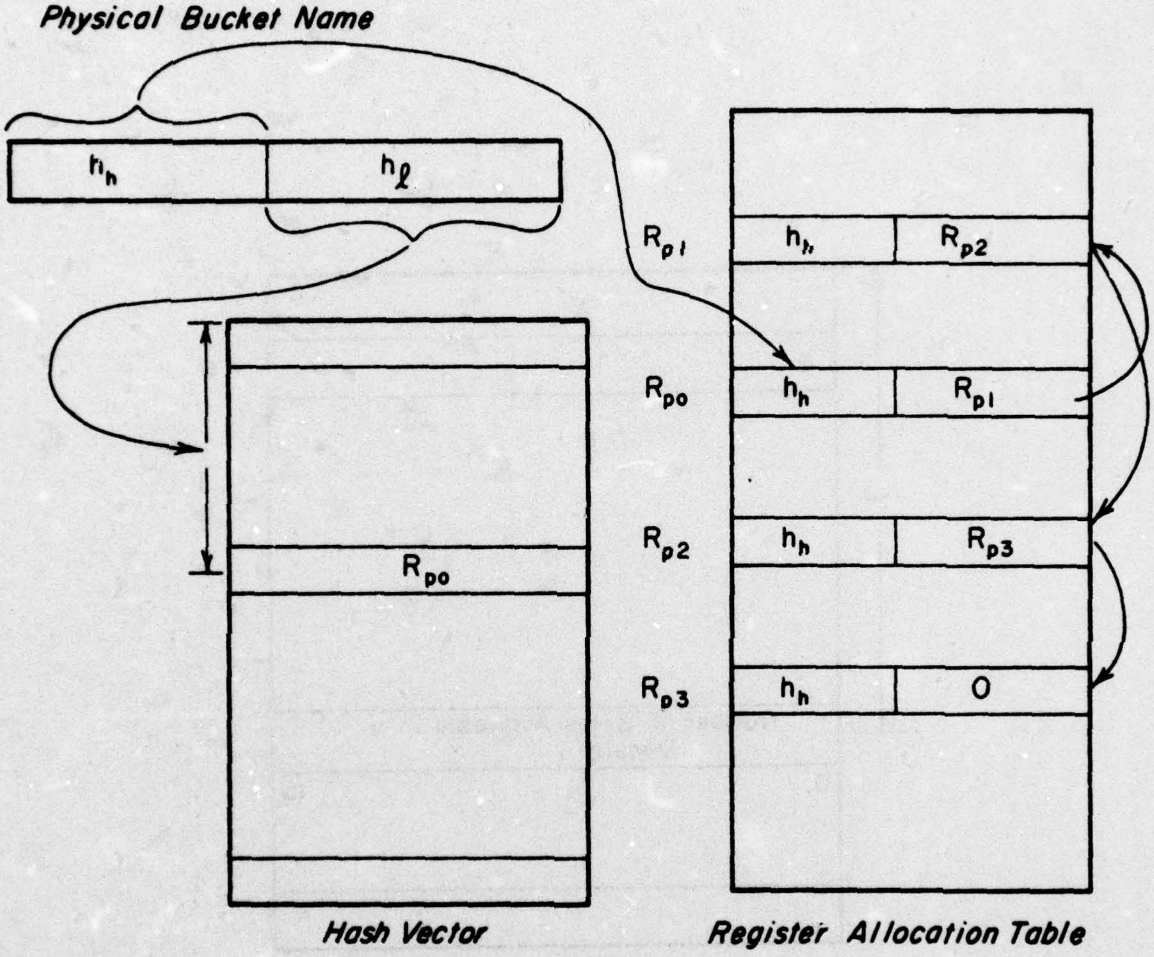


Figure 19. A Case of Multiple Allocation

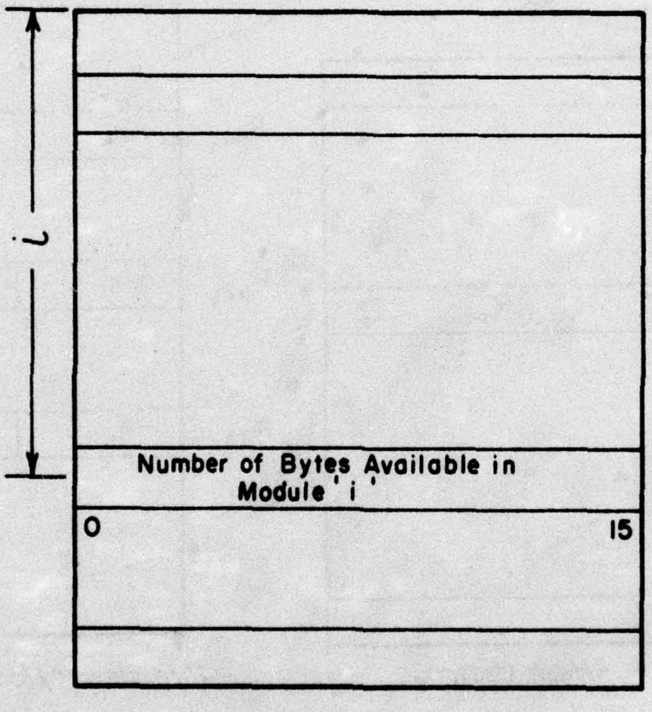


Figure 20. The Module Space Table

C. Sizes and Load Conditions of the Data Structures

The number of entries in the hash vector could assume several possible values with different performance figures. For example, if the number of modules available in a memory unit is, say, n, the number of entries in the hash vector could be 0.25n, 0.5n, n, 2n, 4n, etc. The larger is the vector size, the lower is the probability of two distinct (low order bits of) bucket names hashing to the same entry in the hash vector. The number of bits required in each entry of the hash vector depends on the number of modules in the memory unit attached to a PE. In Table XII, we have tabulated the number of modules required as a function of memory unit size and module size. The memory unit sizes are as they appear in Table V.

In Table XIII, we tabulate the number of bits required by each entry in the hash vector and the number of bits required to address the hash vector in parenthesis for various hash vector sizes. We also tabulate for each hash vector size two performance figures under extremely light and extremely heavy conditions. The two performance figures pertain to the number of entries that must be searched until a particular entry is located or is determined not to be in the table. This quantity is called the number of probes. It has been shown in [10] that for a hash vector using overflow chaining the average number of probes needed to find an entry is C where

$$C = 1 + LF/2$$

and the average number of probes needed to discover that a particular entry is not in the hash table is C' where

$$C' = LF + \exp(-LF)$$

Here, LF is defined as the ratio of the number of allocated entries in the MAT to the number of entries in the hash vector. When the MAT is allocated to 0.9 of its capacity, then we shall consider it heavily loaded. When the table has only 10% of entries allocated, then we shall consider it lightly loaded.

Table XII : No. of modules in a memory unit as a function of the memory unit size and module size

Module Size (K byte) \ Memory Unit Size (K byte)	40	80	160	320	400	800	1600	3200	4000	8000	16000	32000
1	40	80	160	320	400	800	1600	3200	4000	8000	16000	32000
2	20	40	80	160	200	400	800	1600	2000	4000	8000	16000
4	10	20	40	80	100	200	400	800	1000	2000	4000	8000
6	7	14	27	54	67	134	267	534	667	1334	2667	5334
8	5	10	20	40	50	100	200	400	500	1000	2000	4000
10	4	8	16	32	40	80	160	320	400	800	1600	3200
12	4	7	14	27	34	67	134	267	334	667	1334	2667
16	3	5	10	20	25	50	100	200	250	500	1000	2000

Table XIII. Number of bits required in the hash vector and number of bits required to address the hash vector as a function of normalized hash vector size and number of modules.

Number of Modules	Hash Vector Size				
	0.25	0.50	1.00	2.00	4.00
80	7/(5)	7/(6)	7/(7)	7/(8)	7/(9)
160	8/(6)	8/(7)	8/(8)	8/(9)	8/(10)
400	9/(7)	9/(8)	9/(9)	9/(10)	9/(11)
800	10/(8)	10/(9)	10/(10)	10/(11)	10/(12)
1600	11/(9)	11/(10)	11/(11)	11/(12)	11/(13)
3200	12/(10)	12/(11)	12/(12)	12/(13)	12/(14)
8000	13/(11)	13/(12)	13/(13)	13/(14)	13/(15)
16000	14/(12)	14/(13)	14/(14)	14/(15)	14/(16)
32000	15/(13)	15/(14)	15/(15)	15/(16)	--
Heavy Load					
C	2.8	1.9	1.45	1.225	1.113
C'	3.63	1.96	1.30	1.08	1.02
Light Load					
C	1.2	1.1	1.05	1.025	1.013
C'	1.07	1.01	1.004	1.001	1.00

$$\text{Normalized Hash Vector Size} = \frac{\text{Actual Hash Vector Size}}{\text{\# of Register Modules}}$$

(A) Number of Bits required to Address the Hash Vector

$$= \left\lceil \text{Log}_2\,(\text{Number of Modules}) * \begin{pmatrix}\text{Normalized Hash}\\ \text{Vector Size}\end{pmatrix} \right\rceil$$

(B) Number of bits in each entry of the Hash Vector

$$= \left\lceil \text{Log}_2\ (\text{Number of Modules}) \right\rceil$$

(See text for definition of C and C'

Number of bits in each entry in the Module Allocation Table

= Physical Bucket Name Size - A + B

Thus,

$$LF_{\text{heavy load}} = \frac{0.9 \times \text{Total Number of entries in MAT}}{\text{Number of entries in Hash Vector}}$$

$$LF_{\text{light load}} = \frac{0.1 \times \text{Total Number of entries in MAT}}{\text{Number of entries in Hash Vector}}$$

Now, let

$$\text{Normalized Hash Vector Size} = \frac{\text{Number of entries in Hash Vector}}{\text{Total Number of entries in MAT}}$$

Thus,

$$LF_{\text{heavy load}} = \frac{0.9}{\text{Normalized Hash Vector Size}}$$

$$LF_{\text{light load}} = \frac{0.1}{\text{Normalized Hash Vector Size}}$$

From Table XIII, we find that by increasing the relative hash vector size we get better performance in terms of smaller number of probes. However, the percentile increase in performance is not the same each time we double the hash vector size. For example in going from a normalized hash vector size of 0.25 to 0.5 the percentile drop in the number of probes C, is 32%, but in going from 0.5 to 1.0 the percentile drop is only about 23%.

In Table XIX we tabulate memory requirements and performance gain/loss achieved for various sizes of the hash vector. We have arbitrarily chosen the performance of a hash vector whose normalized size is 1 to have unity performance. From the table we see that the performance under heavy load conditions, improves by 28-30% when the hash vector size is made 4 times the size of the MAT. However, the space requirement, in going from a relative hash vector size of 1 to 4, increases from 240 to 480 bytes (a 100% increase) for 80 modules and from 64K to 160K (an increase of 150%) for 16000 modules. Clearly, the performance improvement is not cost-effective. Under light load conditions, the performance improvement is even less cost-effective. In fact, under light load conditions we can get as high as 87.5% to 93% of the

Table XIX. Space requirements in bytes for data structures maintained by each of the memory unit processors.

Number of Modules	Normalized Hash Vector Size				
	0.25	0.5	1.0	2.0	4.0
80	20 + 240	40 + 240	80 + 160	160 + 160	320 + 160
160	40 + 480	80 + 480	160 + 320	320 + 320	640 + 320
400	200 + 1200	400 + 1200	800 + 800	1600 + 800	3200 + 800
800	400 + 2400	800 + 2400	1600 + 1600	3200 + 1600	3200 + 1600
1600	800 + 4800	1600 + 4800	3200 + 3200	6400 + 3200	12800 + 3200
3200	1600 + 9600	3200 + 9600	6400 + 6400	12800 + 6400	25600 + 6400
8000	4000 + 24000	8000 + 24000	16000 + 16000	32000 + 16000	64000 + 16000
16000	8000 + 48000	16000 + 48000	32K + 32K	64K + 32K	128K + 32K
32000	16000 + 96K	32000 + 96K	64K + 64K	218K + 64K	--
Heavy Load					
P	0.517	0.763	1.00	1.18	1.30
P'	0.359	0.656	1.00	1.209	1.28
Light Load					
P	0.875	0.95	1.00	1.024	1.037
P'	0.93	0.99	1.00	1.002	1.004

Space Required by Hash Vector = (# of Modules)*(Normalized Hash Vector Size)

*(# of Bytes required per entry in Hash Vector)

Space Required by Module Allocation Table = (# of Modules)

*(# of Bytes per entry)

Performance $P = \frac{1/C}{1/C_1}$ where C_1 = # of probes when normalized hash vector size is 1.

Performance $P' = \frac{1/C'}{1/C'_1}$ where C'_1 = # of probes (for no match) when normalized hash vector is size 1.

performance with a hash vector which is one fourth the size of the unity performance hash vector. But under heavy load conditions, small vector sizes can really hurt performance (51%-35.9%). Thus the choice of the relative vector size narrows down to between 0.5 and 2. We shall not attempt to narrow down the range any further at this point, since a more detailed analysis in terms of memory cost and overall performance requirements is needed.

D. Logic of the PE

The orders issued by the SMC to the PEs are as follows:

1) Find the number of modules in use for the physical bucket named m.
2) Find the total number of modules available.
3) Look up physical bucket m for a transformed keyword whose low order bits have the value T_ℓ.
4) Retrieve from physical bucket m the directory entry of a transformed keyword whose low order bits have the value T_ℓ.
5) Retrieve all index terms from the directory entries in the physical bucket m.
6) Delete from physical bucket m the index term i for the transformed keyword whose low order bits have the value T_ℓ.
7) Insert into module j of physical bucket m the index term (i,(q)) for the transformed keyword whose low order bits have the value T_ℓ.
8) Create an entry in physical bucket m for the transformed (security/clustering) keyword whose low order bits have the value T_ℓ and the index is (i,(q)).

For each of the above orders there is one corresponding algorithm which is executed by the PEs. The first two orders are used by the SMC to ensure that modules of a physical bucket are uniformly distributed over all the memory units.

ALGORITHM A: **To find the number of modules in use with a bucket and determine the space available in each of the modules. (Result of the algorithm is available in an array called RESULT.)**

Input Arguments: Bucket number m

Step 1: r ← 0

Step 2: Use the appropriate number of bits in m to index the hash vector table (see Figure 18). If the entry is empty, go to Step 7; else, extract the pointer from the entry. Call it R_p.

Step 3: If the high order bits in the R_p-th entry of MAT matches the high order bits of m, then go to Step 4; else, go to Step 5.

Step 4: r←r+1; RESULT[r]←R_p,MST[R_p].

Step 5: R_p←MAT[R_p]. (i.e., extract the pointer to the next entry in MAT)

Step 6: If R_p=0, then go to Step 7; else, go to Step 3.

Step 7: RESULT[0]←r. Terminate.

Response: RESULT[0] contains the number of modules allocated to a physical bucket. RESULT[1] through RESULT[r] indicate the space availability in each of the modules.

ALGORITHM B: To find the number of modules available in the memory unit attached to a PE. (Each PE maintains a count called COUNTER to keep track of the number of modules available. This counter is manipulated by other algorithms).

Step 1: RESULT[0]←COUNTER. Terminate.

ALGORITHM C: To look up bucket m for a transformed keywork whose low order bits have the value T_ℓ.

Input Arguments: Bucket name m, and low order bits T_ℓ.

Step 1: SEARCHFLAG←false

Step 2: Call Algorithm A to obtain the set of modules allocated to bucket m.

Step 3: If RESULT[0]=0, then go to Step 8; else, go to Step 4.

Step 4: r←RESULT[0]; j←1;

Step 5: Search the module given by RESULT[j] for a directory entry whose transformation field matches the argument T_ℓ. If match occurs, then go to Step 7; else, go to Step 6.

Step 6: j←j+1; if j≤r, then go to Step 5; else, go to Step 8.

Step 7: SEARCHFLAG←'true';RESULT[1]←RESULT[j]
RESULT[2]←MST[RESULT [j]]

Step 8: RESULT[0]←SEARCHFLAG; Terminate.

<u>Note</u>: In the above algorithm Step 5 is a crucial one, as it is responsible for the search operation. Searching can be done on the fly or on a buffered basis depending on processor speed and readout times for a module

ALGORITHM D: To retrieve from physical bucket in the directory entry of a transformed keyword whose low order bits is T_ℓ.

Input Arguments: Bucket name m, low order bits of transformed value T_ℓ.

Step 1: SEARCHFLAG←'false'

Step 2: Use the hash vector and module allocation table (Figure 18) to obtain the module number of the next module allocated to m. Call it j.

Step 3: If Step 2 does not yield a module name go to Step 6.

Step 4: Search module j for the directory entry of a transformed keyword whose low order bits are given by T_ℓ. If a match is found, go to Step 5; else, go to Step 2.

Step 5: SEARCHFLAG←'true'; store directory entry in RESULT array.

Step 6: RESULT[0]←SEARCHFLAG: Terminate.

Response: Directory entry in RESULT array (if found).

ALGORITHM E: To retrieve all the directory entries in bucket m.

Input Arguments: Bucket name m

Step 1: SEARCHFLAG←'false'

Step 2: Use the hash vector and module allocation table (Figure 18) to obtain the number of the next module allocated to m. Call it j.

Step 3: If Step 2 does not yield a module name, go to Step 6.

Step 4: Readout module j and store its contents in RESULT array.

Step 5: SEARCHFLAG←'true'. Go to Step 2.

Step 6: RESULT[0]← SEARCHFLAG ; terminate.

Response: All directory entries in RESULT array.

ALGORITHM F: To delete from physical bucket on the index term i for the transformed keyword whose low order bits have the value T_ℓ.

Input Arguments: module m, index i, transformed value's low order bit T_ℓ.

Step 1: Call Algorithm A to find all modules allocated to bucket m.

Step 2: If RESULT[0]=0, then terminate.

Step 3: j←1; r←RESULT[0].

Step 4: Search module identified by RESULT[j] for the directory entry of a transformed keyword whose low order bits have the value T_ℓ. If a match occurs, go to Step 6; else, go to Step 5.

Step 5: j←j + 1; if j≤r, go to Step 4; else, terminate.

Step 6: [Match has occurred.] Look for index term i within the entry. If found, delete it. If i is the last index in the entry, then go to Step 7; else, go to Step 9.

Step 7: [Entry to be deleted.] EFLAG←1. Reduce count of number of entries in module by 1. If number of entries is zero, then go to Step 8; else, go to Step 9.

Step 8: [Module to be freed.] Use the hash vector to trace the chain of pointers for the bucket m. When the j-th entry is accessed, update the entry which is pointed to the j-th entry (called the predecessor) by copying the contents of the j-th entry into the predecessor. Link up the j-th entry into the AVAIL list. [The module is now available for reallocation.]

Step 9: Increase MST[j] by the length of index term just deleted. If EFLAG = 1, then increase MST[j] by the number of bytes occupied by the transformed value and index term count.
EFLAG←0. Terminate

Response: None

ALGORITHM G: To insert into module j of physical bucket m, the index term (i, (q)) for the transformed keyword whose low order bits have the value T_ℓ.

Input Argument: Bucket name m, module number j, index term (i, (q)) and transformed value T_ℓ.

A call to this algorithm is always preceded by a call to Algorithm C which looks up the bucket containing the keyword entry into which the index term is to be inserted. Recall that Algorithm C returns the name of the module containing the entry if the lookup is successful, i.e., SEARCHFLAG = 'true'.

Step 1: Read module j into buffer if it not already in the buffer. [It is quite likely that the information in module j is already in the input buffer, since Algorithm C searched the module before the invocation of this algorithm.]

Step 2: Scan index terms of the entry for the transformed value T_ℓ. If i already exists, then terminate.

Step 3: Move the directory entries of other keywords in the buffer to make space for the current index term (i, (q)). [There will always be enough room, since the SMC does not issue this order if there is not enough space in the module.]

Step 4: Update MST[j] to reflect the space occupied by the new index term. Terminate.

Response: None

ALGORITHM H: Create an entry in bucket m with index term (i, (q)) for the (security/clustering) keyword whose low order bits have the transformed value T_ℓ.

Input Arguments: bucket m. Index term (i, (q)) and T_ℓ.

Step 1: Call Algorithm A to find the set of modules already allocated to bucket m. If *RESULT[0]* = 'false', go to Step 3.

Step 2: Find from the set of modules identified by RESULT[1] through RESULT [r], the module k which has the largest space available and exceeds the space required by the entry. If no such module exists, go to Step 3; else, go to Step 4.

Step 3: [Allocate new module] From the AVAIL list, pick up the first available module k, and link it up in the chain emanating from the hash vector entry for m. (see Figures 18 & 19). Go to Step 5.

Step 4: Read the module k chosen in Step 2.

Step 5: Place the argument entry in the first available location indicated by MST[k]. Update MST[k] to reflect the space occupied by the new entry.

Step 6: Write back the module k. Terminate.

Response: None

E. PEs and the Structure Memory Controller

The above algorithms have been carefully tailored to eliminate much of the decision making tasks from the logic of the PEs. The algorithms are only capable of transmitting the information to the SMC which makes decisions. These decisions pertain to uniform distribution of keyword directory entries and taking care of overflows in modules, etc. For example, SMC, before making an insert request, orders the PEs to see if the keyword already has an entry, and to determine the amount of space available in the module containing the entry. Based on these two pieces of information, the SMC can make a decision on where to insert the index term. We shall see more of this decision making procedure in the algorithms of the SMC.

Most of the algorithms need working space (for insertions, deletions, etc.) The working space required is of the order of the size of a module. In some instances, particularly in the "Retrieve All" order (Algorithm E) the amount of data retrieved may exceed the work space available. In such cases, the retrieval is done in burst mode, i.e., when the work space fills up, it is transmitted to the SMC before more retrieval is attempted.

3.3 The Structure Memory Controller (SMC)

The SMC is responsible for carrying out the following functions:

- Transformation of logical bucket name to physical bucket name
- Controlling the bucket memory system (BMS)
- Controlling the structure memory information processor (SMIP)
- Maintaining the look-aside buffer memory system
- Executing the following DBCCP commands
  - Retrieve Index term for a keyword predicate
  - Retrieve with count the index terms for a (Security/ clustering) keyword
  - Insert index term for a keyword
  - Delete index term for a keyword
  - Reset

The algorithms executed by the SMC are sequential in nature; parallelism is induced by the simultaneous invocation of the PEs of the BMS. The critical requirement of the SMC hardware is that it should be able to move information in and out of the SM and make decisions at a rate which is comparable to the rate of flow of information from the PEs.

3.3.1 Transformation of Logical Bucket Name to Physical Bucket Name

The rationale for transforming logical bucket names to physical bucket names has already been put forth in an earlier section. Here we discuss its implementation. In Figure 21, we detail the basic data structure required to implement the mapping. The structures are similar to those involved in mapping a physical bucket into a set of module names. The main difference, here, is that for a given logical bucket name there is only one physical bucket name.

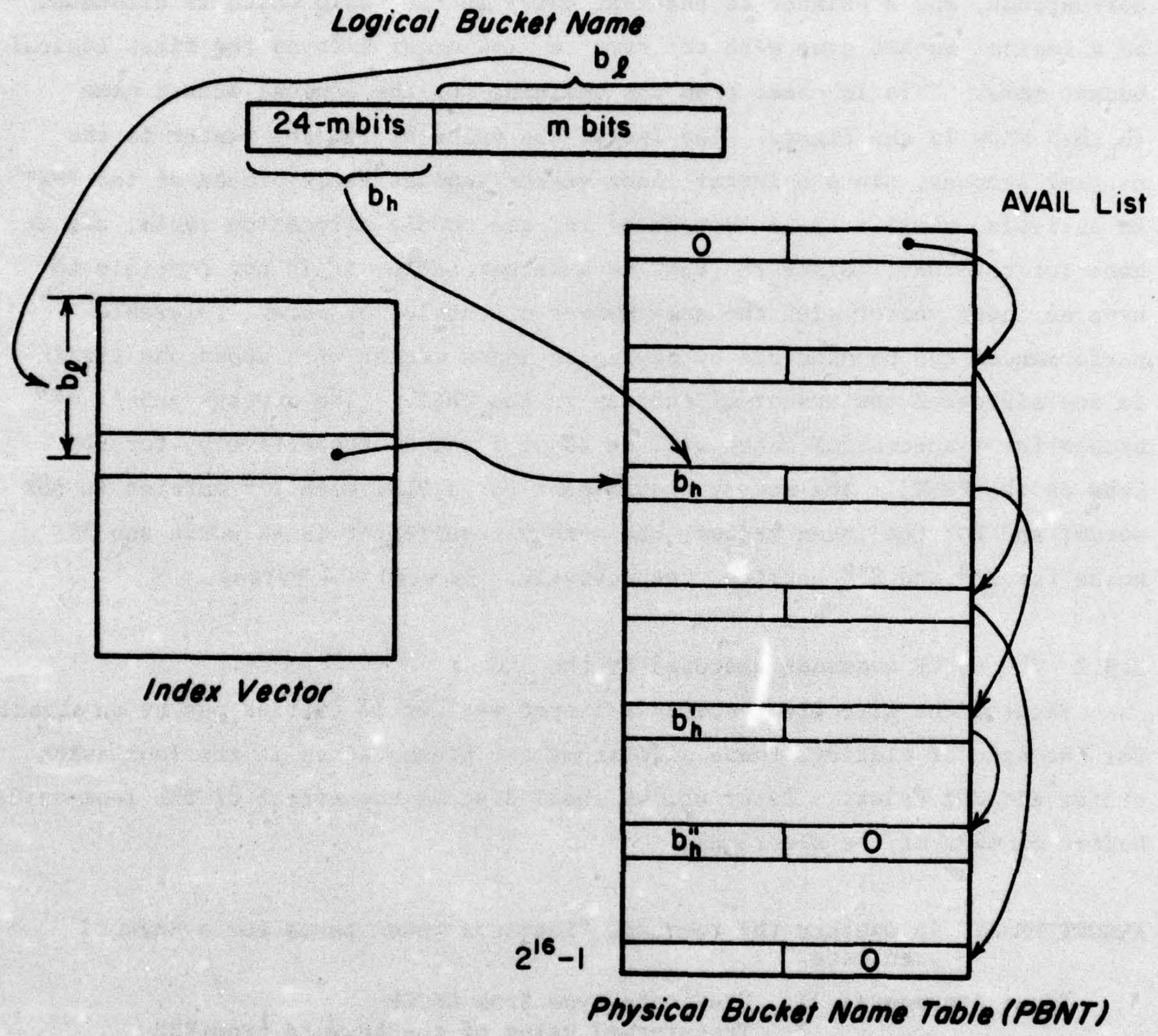


Figure 21. Logical to Physical Bucket Transformation

The m low order bit of the logical name is used to access the index vector. The corresponding entry in the index vector contains a pointer to the physical bucket name known to the SMC. Thus, (since we had assumed a 16 bit physical bucket name in earlier sections), we need 2^{16} entries. Each entry has the (24-m) high order bits of the logical bucket name to which the physical bucket corresponds, and a pointer to the next entry in the table which is allocated to a logical bucket name with the same m low order bits as the first logical bucket name. This is clear from the chaining for the logical bucket name (b_h, b_ℓ) show in the figure. The larger the value of m, the faster is the mapping process, since a larger index vector implies fewer probes of the PBNT. An analysis, similar to one conducted for the module allocation table, can be made for the PBNT. Since the PBNT is a larger table, it is not feasible to have an index vector with the same number of entries or more. Tolerable performances can be obtained by having an index vector with about one eighth to one sixteenth the number of entries in the PBNT. [The average number of probes for a successful match will be about 5 and 9, respectively, for 90% load on the PBNT]. The memory requirement for a PBNT with 2^{16} entries is 65K words; and for the index vector, the memory requirement is 4K words and 8K words for 2^{13} and 2^{14} entries, respectively. [a word = 4 bytes].

3.3.2 The DBCCP commands executed by the SMC

Each of the five DBCCP commands listed earlier is carried out by an algorithm. For the sake of clarity, these algorithms are presented as if the look-aside buffer did not exist. Later on, we shall discuss the effect of the look-aside buffer on each of the algorithms.

ALGORITHM A: To execute the command, "Retrieve index terms for a keyword predicate."

Input Arguments:
1. Predicate type from DBCCP
2. Transformed value of the keyword from KXU

[Note that the DPCCP sends the predicate type $(=,\neq,\leq,<,\geq)$ along with the command, while the keyword is sent via KXU where it is transformed as described in Section 2.]

Step 1: Obtain the transformed keyword value T(K) from the KXU.

Step 2: If the predicate is '=' or '≠', then go to Step 3; else, go to Step 7.

Step 3: Use the index vector and PBNT to obtain the name of the physical bucket corresponding to the logical bucket name. Call it m.

Step 4: Issue the order to the PEs, "Retrieve from physical bucket m, the directory entry of a transformed keyword whose low order bits have the value $T_\ell(K)$".

Step 5: If the predicate is '=', then send the retrieved indices to the SMIP for futher processing, set DFLAG 0, and terminate. Else, go to Step 6.

Step 6: [The predicate is '≠'] Send the retrieved indices to the SMIP for further processing, set DFLAG = 1, and terminate. [See Section 4.1 for the function of DFLAG].

Step 7: [Predicate is one of '≥', '>', '≤', '<']. If predicate is of type '≤' or '<', then go to Step 12; else, go to Step 8.

Step 8: [Predicate is either '≥' or '>']. From the logical bucket name and in the transformed value, obtain the partition number p. If the predicate is '>', then $p \leftarrow p + 1$.

Step 9: Use the index vector and PBNT to determine the names of the physical bucket for all the logical bucket names whose partition numbers are equal to or greater than p. Call this set of physical bucket names I.

Step 10: For each physical bucket name m in I, issue the order, "Retrieve from physical bucket m, all the index terms of all the directory entries."

Step 11: Send the retrieved indices to the SMIP for further processing with DFLAG = 0. [See Section 4.2 for details.] Terminate.

Step 12: [Predicate is either '≤' or '<']. From the logical bucket name and in transformed value, obtain the partition number p. If the predicate is '<', then $p \leftarrow p-1$.

Step 13. Use the index vector and PBNT to determine the names of the physical buckets for all the logical bucket names whose partition numbers are equal to or less than p. Call this set of physical bucket names I. Go to Step 10.

Notes on Algorithm A: This algorithm provides a limited facility for processing keyword predicates which are of the type '≤', '<', '≥' and '>'. The limitations arise from the manner in which keyword directory entries are stored in the BMS. The reader will recall from Section 2, that the partition number in the logical bucket name associated with a keyword, reflects a range (of values) within which the keyword happens to be. A range has a low extreme V_ℓ and an upper extreme V_h. Algorithm A will perform correctly under the following conditions: The predicate '<' must be associated with the lower extreme V_ℓ of some range. The predicate '≤' must be associated with the upper extreme value V_h of some range. The predicate '>' must be associated with the upper extreme value V_h of some range. The predicate '≥' must be associated with a lower extreme value V_ℓ of some range. These four conditions are not nearly as restrictive as they seem to be, since in practice range searches involving these predicates are carried over ranges which can be predicted in advance, and therefore can be associated exactly with partitions. If the second or fourth condition is violated, more indices will be retrieved than necessary. If the first or third condition is violated, some relevant indices may not be retrieved.

ALGORITHM B: To retrieve with count the index terms for the (security/clustering) keyword.

Input Arguments: 1. Count value from DBCCP
2. Transformed value of the keyword K from KXU

[This algorithm is used to find out the name of the cluster or security atom to which a record belongs. The predicate associated with this algorithm is always '='. The count value is the number of clustering/security keywords present in a record belonging to the cluster or security atom.]

Step 1: Obtain the transformed keyword value T(K) from the KXU.

Step 2: Use the index vector and PBNT to obtain the name of the physical bucket corresponding to the logical name in T(K). Call it m.

Step 3: Issue the order to the PEs "Retrieve from physical bucket m, the directory entry of a transformed keyword whose low order bits have the value $T_\ell(K)$".

Step 4: For each of the index term retrieved in Step 3, compare the associated count fields against the argument count value supplied by the DBCCP. Form the set S of index terms for which the equality operation is successful.

Step 5: Send the set S to the SMIP with DFLAG = 0. Terminate.

ALGORITHM C: Insert index term for a keyword.

Input Arguments: 1. Index term from DBCCP.
2. Count value from DBCCP if applicable [only in case of security or clustering keywords].
3. Transformed value of keyword from KXU.

Step 1: Obtain the transformed keyword value T(K) from the KXU.

Step 2: Use the index vector and PBNT to obtain the name of the physical bucket corresponding to the logical bucket name. If the PBNT fails to yield the physical bucket name, then allocate a physical bucket from the AVAIL list. Call the bucket name m. If bucket m is newly allocated, skip to Step 13.

Step 3: Issue the order, "Look up bucket m, for the directory entry of the transformed keyword whose low order bits have the value T_ℓ" to all the PEs.

Step 4: If the SEARCHFLAG of one or more PEs is 'true', then go to Step 5; else, go to Step 7.

Step 5: Determine the PE whose module has the largest space available. Call it j.

Step 6: If the space in j is greater than that required by the index term (and its count if applicable), then issue the order, "Insert into physical bucket m, module j, the index term (i, (q)) for the keyword whose low order bits have the transformed value T_ℓ and terminate. If space in j is not enough, go to Step 7.

Step 7: [Control comes to this step if no PE reports the existence of the directory entry of the concerned keyword in Step 3, or if the space in any of the modules is insufficient to contain the index term as determined in Step 6]. Issue the order, "Find the number of modules in use for physical bucket with name m, to all PEs.

Step 8: From the response to the order in Step 7, choose the set of PEs which have no module allocated to m and call this set Z. If the set Z is empty, go to Step 12. $[Z = \{p_1, p_2, \ldots, p_r\}]$.

Step 9: To each of the PEs identified in Z, issue the order, "Find the number of modules available." Call the response set Z' $[Z' = \{(p_1,n_1), (p_2,n_2), \ldots, (P_r,n_r)\}]$.

Step 10: From Z' choose the one PE p' which has the largest number n' of modules available.

Step 11: Issue to p', the order, "Create an entry in bucket m for the transformed (security/clustering) keyword whose low order bits have the value T_ℓ and the index is (i, (q)). Terminate.

Step 12: [Control comes here when all PEs have modules allocated to m]. Choose the PE(s) which has (have) the smallest number of modules allocated to m. If the choice is reduced to one PE p' go to Step 11; else, let $Z \equiv \{p|p$ has the smallest number of modules allocated to m$\}$. Go to Step 9.

Step 13: [Control comes here when bucket m has been allocated in Step 2, and therefore the BMS has no entries in m]. Let Z = {all PEs}, go to Step 9.

ALGORITHM D: To delete an index term for a keyword

Input Argument: 1. Index term i from DBCCP
2. Transformed keyword value T(K) from KXU

Step 1: Obtain the transformed keyword value T(K) from KXU

Step 2: Use the index vector and PBNT to obtain the name of the physical bucket corresponding to the logical bucket. Call this name m.

Step 3: Broadcast to all PEs the order, "Delete from physical bucket m the index term i for the transformed keyword whose low order bits have the value $T_\ell(K)$".

Step 4: [Determine if this was the last entry in bucket m]. Broadcast to all PEs the order, "Find number of modules attached to bucket m."

Step 5: If all the PEs respond to the order in Step 4, with zero as the number of modules attached to the bucket m, then m can be reused; else, terminate.

Step 6: [Physical bucket m in freed]. Place the entry for m in PBNT in the AVAIL list. Update the chain for the logical bucket name in the index vector. Terminate.

ALGORITHM E: To execute the command 'reset.'

Input Argument: None

[This algorithm is primarily intended to signal the SMC that a string of retrieve commands is ended. The DBCCP always presents the SM with a string of retrieve commands corresponding to a query conjunct or a set of keywords in a record].

Step 1: Issue a command to the SMIP to retrieve all valid data units from its memory and then clear the SMIP memory. Terminate.

3.3.3. The Relationship of SMC, SMIP, DBCCP, and KXU.

We have seen data structures maintained by and algorithms executed by the SMC. These algorithms concern themselves with the retrieval (or insertion) of information from (into) the array of PEs. Information so retrieved is passed on to the SMIP; and the information to be placed in the SM is received from the KXU and DBCCP. The overall flow of the information through the SMC is schematically shown in Figure 22.

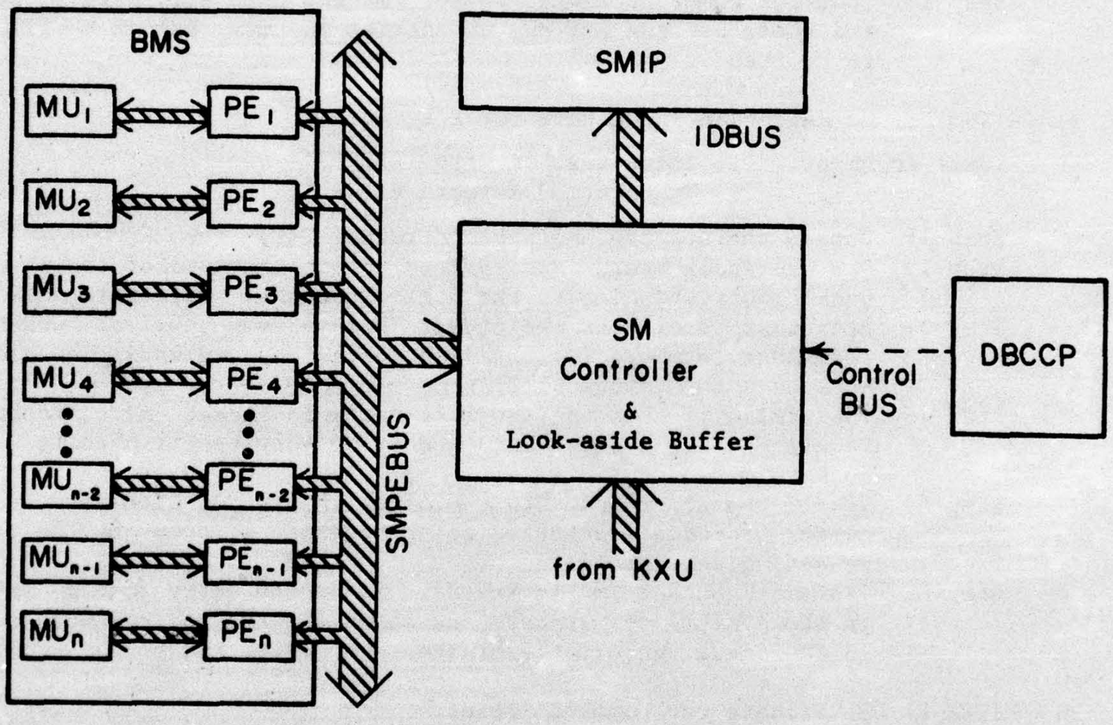


Figure 22. Flow of Information through SMC

The PEs of the BMS can be selectively activated by the SMC using a mask register. There is one bit in the mask register for each PE. An order issued by the SMC is received by all the PEs, but only those which have the corresponding mask bit turned on, carry out the order. The bits in the mask register can be set (or reset) by the SMC on an individual basis. Data transfer between the PE and the SMC takes place over a data bus called the SMPEBUS (see Figure 22). Such data transfer is always in response to or a part of orders broadcast by the SMC. Thus, communication between the SMC and the PEs is in a master-slave relationship, with the SMC assuming the master role and the PEs assuming the slave role. The design of the control function and the SMPEBUS is conventional and, therefore, will not be described here.

The relationship between the SMC and the SMIP is also a master-slave one, with the SMC assuming the master role. Whenever the SMC executes a retrieve command issued by the DBCCP, the response of that command is sent to the SMIP as soon as the SMIP is ready to accept the data. Data transfer is always from the SMC to the SMIP in burst mode. When the reset command is issued by the DBCCP, the SMC informs that the current string of data is completed, and that the SMIP should send all valid data units to the index translation unit (IXU). After retrieval, the SMIP clears its memory and readies itself for the next data string.

3.4 The Look-Aside Buffer Memory (LABM)

Before describing the LABM, we introduce a few notations which will be used rather frequently in the ensuing discussion. A command received by the SMC from the DBCCP will be called an input command, while a command already present in the LABM will be called a buffer command. A command has up to three arguments: a transformed keyword value T, an index term I, and a count C. The subscript i will be used to identify input command arguments and the subscript b will be used to identify buffer command arguments.

3.4.1 Design and Implementation of the LABM

The look-aside buffer memory is divided into two components: an associative memory (AM) and a random access memory (RAM). The AM has n data cells each of which has a 24-bit search key and a pointer to the random access memory. The search key is the logical bucket name specified in the transformed keyword value of one or more buffer commands. The pointer directs to a list of commands in the RAM whose transformed keyword value arguments are all referring to the same logical bucket name in the corresponding search key. (See Figure 23a). The AM is searched on the basis of a logical bucket name occurring in T_1. The response set of the AM contains the pointers to those data cells whose search keys satisfy the search criterion. The search criterion may be '=', '≥', '>', '≤', or '<'. The RAM which holds the command is divided into m entries each of which can hold one command. Each entry has six fields (see Figure 23b).

FIELD 1: List Pointer - This field points to the next command entry (in the RAM) whose transformed keyword value refers to the same logical bucket name as the command in the current entry. This field is $[\log_2 m]$ bits long.

FIELD 2: Queue Pointer - This field is used to maintain a FIFO discipline among all the commands awaiting execution in the LABM. The field points to the update command which arrived next in the time sequence. It requires $[\log_2 m]$ bits.

FIELD 3: Order Code - This field is used to indicate whether the command is an insert or a delete command and is one bit long. [Order Code = 1 for insertion; = 0 for deletion.]

FIELD 4: Transformation Value - This field holds the 48 bits of the transformed value associated with the command.

FIELD 5: Index Terms - This field holds the 28-bits index term which is to be deleted or inserted.

FIELD 6: Count - This field is used to indicate the number of security (or clustering) keywords in the security atom (or cluster) identified in the index term in Field 5.

3.4.2 Effects of the LABM on SMC Command Execution

Algorithms executed by SMC are affected by the presence of the LABM.

A. Delete Commands (See Algorithm D in Section 3.3.2) - When an delete command is received by the SMC, the SMC orders the AM of the LABM to search for the logical bucket name contained in T_1. If the response set is empty, then a command entry is created for the command in the RAM and a data cell is created in the AM for the logical bucket name in T_1. The pointer in the data

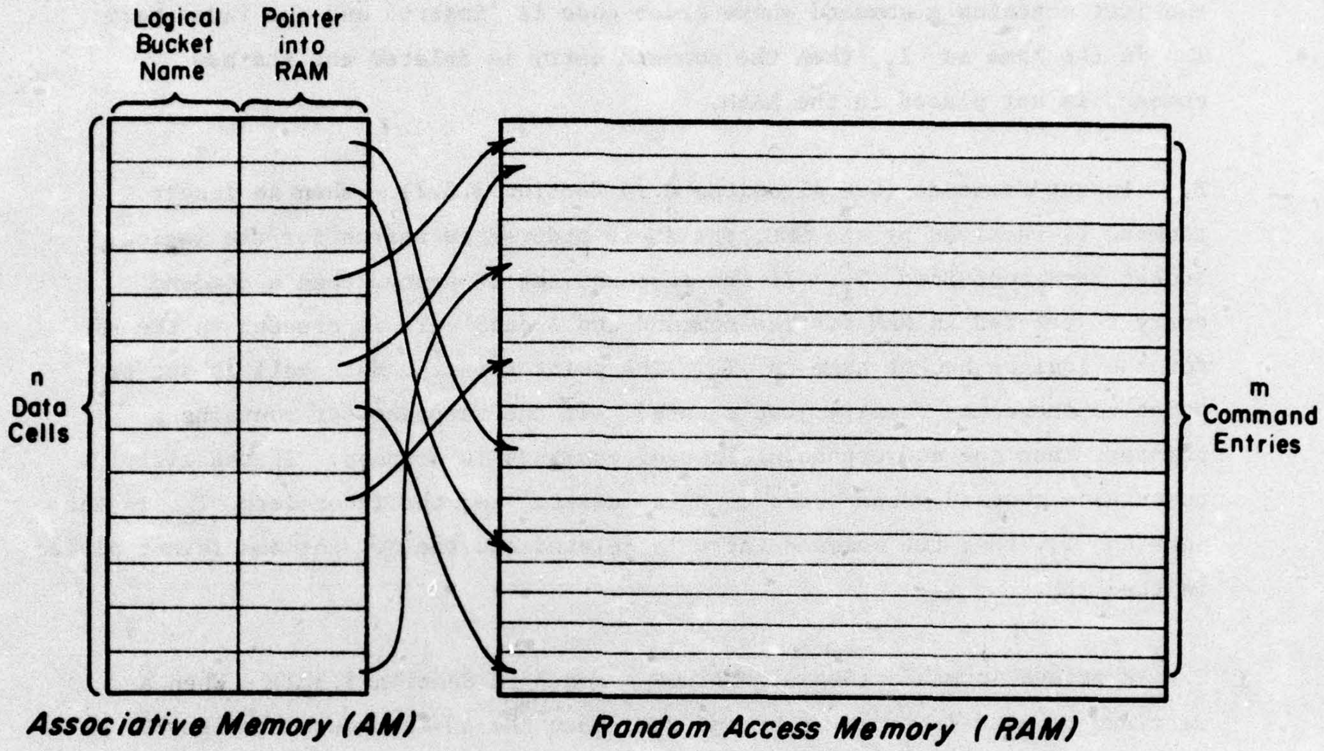


Figure 23a. Organization of Look-Aside Buffer Memory (LABM)

List Pionter	Queue Pionter	Order Code	Transformation Value	Index Term	Count
Field-1	Field-2	Field-3	Field-4	Field-5	Field-6

Figure 23b. Format of the Command Entry

cell is set to point to the command entry just created. If the response set contains a pointer, then the corresponding list of commands is scanned. If the list contains a command whose order code is 'insert' and the index term I_b is the same as I_i, then the command entry is deleted and the new command is not placed in the LABM.

B. Insert Commands (See Algorithm C in Section 3.3.2) - When an insert command is received by the SMC, the AM is ordered to search for the logical bucket name contained T_i. If the response set is empty, then a command entry is created in RAM for the command and a data cell is created in the AM for the logical bucket name in T_i. The pointer in the data cell is set to point to the command entry just created. If the response set contains a pointer, then the corresponding list of commands is scanned. If the list contains a command whose order code is 'delete' and the index term I_b is the same as I_i, then the command entry is deleted and the new command is not placed in the LABM.

C. Retrieve Commands (See Algorithms A and B in Section 3.3.2) - When a retrieve command is received by the SMC, then the AM is ordered to search for those logical bucket names that satisfy the command's predicate with respect to the logical bucket name in T_i. If the response set is not empty, then the lists pointed to by the pointers in the response set are scanned for 'insert' commands. The index terms I_b of such commands are placed in the retrieval set. Either Algorithm A or B in Section 3.3.2 is then executed. As a result of the execution, the retrieval set is augmented. At this point, the AM is again queried for logical bucket names satisfying the retrieval predicate. The lists (if any) are scanned for 'delete' commands. The index terms (I_b) of such commands are removed from the retrieval set, if they occur in the retrieval set.

The AM has a very small number of data cells, typically, under 64. The exact number will be determined by the performance improvement achieved for a certain number of data cells in relation to the cost of implementing the full associativity. The number of command entries that can be stored in the RAM should be larger than the number of data cells in the AM by a factor governed by the average number of commands per logical bucket name. In many instances, we do not expect retrieval sets to be affected by commands in the LABM. This implies that the response set of the AM will be emtpy for many search orders.

Thus, the availability of an AM enables us to avoid fruitless scans of the command queue for each and every retrieval command. In the absence of an AM, such scans become inevitable and may result in performance degradation.

When the AM is full, (i.e., no data cells are vacant), the LABM is considered full, although there may be a few vacant command entries in the RAM. Such a condition forces the SMC to initiate execution of the pending update commands on a FIFO basis. Each of the commands will require the execution of either Algorithm C or D in Section 3.3.2. The execution of LABM commands may also take place if the SMC has not received a retrieve command for a length of time known as the time-out period.

4. THE STRUCTURE MEMORY INFORMATION PROCESSOR (SMIP)

This section presents the part of the DBC known as the structure memory information processor (SMIP). The SMIP is a processor which performs intersection on sets of index terms provided for by the SM of the DBC. The architecture of SMIP is presented in three stages: First, we describe the SMIP as a logical machine. Next we discuss an implementation of the SMIP involving hashing. Finally, the physical realization of the algorithms are discussed in some depth.

4.1 Logical Description of the SMIP

As shown in Figure 24, the SMIP maintains an intermediate set which is subsequently modified by the argument sets in the course of intersection. The argument sets are supplied by the SM consisting of sets of index terms. The intermediate set is designated SW and consists of couples (m,d) called SMIP data units. The first part m of the couple is called the key and the second part d is called the data. The manner in which this intermediate set is manipulated is determined by the state of the SMIP and the command received from the SM.

The concept of a partitioned content addressable memory (PCAM), used to implement the SM,is also used here to realize the intermediate set. As shown in figure 24, the intermediate set is partitioned into ~~a~~ M subsets, each of which contains one or more data units. As in the case of the SM, searching is the most important operation carried out in the SMIP. Index terms in an argument set are used as search keys to determine which one of the partitions has to be searched for the existence of a data unit with a corresponding matching key.

The SMIP can be in one of three states. These three states are known as the initial, the active and the retrieval state. The SMIP is in the initial state when it is ready to accept the first argument set from the SM. The SMIP is in the active state when it has already processed one or more argument sets and has formed an intermediate set. The SMIP is in the retrieval state when the last of the argument sets has been received and the SMIP is in the process of retrieving valid data units and sending them to the IXU. We shall see later what we mean by valid data units.

There are two kinds of SMIP commands. The first kind of SMIP command is represented by SMIP<m,g> where m is a key and g is a manipulation

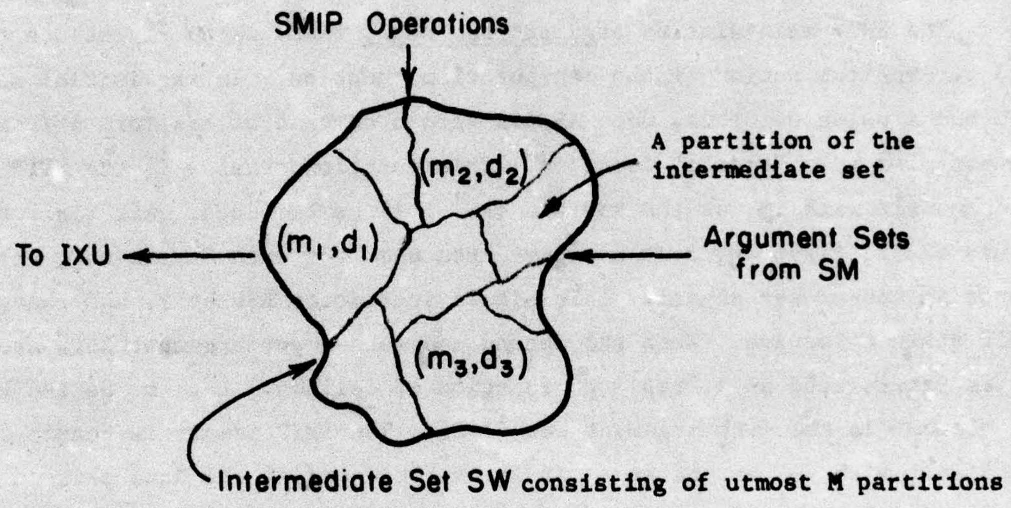

SMIP Operations
A partition of the intermediate set
(m_2,d_2)
(m_1,d_1)
To IXU
Argument Sets from SM
(m_3,d_3)
Intermediate Set SW consisting of utmost M partitions

Figure 24. Conceptual Model of SMIP

function. The manipulation function can do one of two things depending on the state of the SMIP. If the SMIP is in the initial state, then g is interpreted as a "create" function which creates a SMIP data unit for m with the data part d=1. When the SMIP is in the active state, g is interpreted as a "replace" function which may modify the data part of an existing SMIP data unit with key m under certain conditions or do nothing if a SMIP data unit with key m is not found. The second kind of SMIP command is represented by SMIP<retrieve> which signals the end of the intersection operation and requests the SMIP hardware to retrieve all valid data units.

The SMIP maintains an argument set count (ASC) which represents states and governs the action of the manipulation function. In the initial state, the ASC has a value 0. Thus, when ASC is zero a command of the form SMIP<m,g> is interpreted as a "create" function. This function creates in the SMIP memory a data unit with m as the key and (ASC + 1) as the data. After all the elements of the first argument set have been used for such "creation", the SM sends an end-of-set signal. This signal increments ASC by 1, and changes the SMIP state to active. When the second and subsequent argument sets are received, g is interpreted as a "replace" function as follows: Let m be the key of an element in the i-th argument set (i>1). The SMIP memory is searched for a data unit with m as the key. If it is found and if the data part d is equal to ASC, then d is replaced by (d + 1). If the search does not succeed in finding a data unit with m as the key, then no action is taken. When all the elements of the i-th argument set have been processed in this way, ASC is incremented by 1 and the SMIP remains in the active state to process the next argument set. After all arguments sets have been dispatched by the SM, the SM sends the command SMIP<retrieve>. This command changes the SMIP state to retrieve. In the retrieve state, the data parts of the data units in the SMIP memory are compared to the value of ASC. Those data units whose data part d equal the ASC are retrieved and sent to the IXU. Such data units are known as valid data units. After the retrieval process is completed, the SMIP returns to its initial state by setting ASC to 0.

For simplicity, we have not touched upon the role of the DFLAG (first mentioned in Section 3.3.2) which accompanies every argument set except the first. We shall now briefly describe the effect of the DFLAG. When the DFLAG = 0, the SMIP behaves exactly as reported above. However, when the DFLAG = 1, the g function acts as a delete function. The keys of every element in an argument set with DFLAG = 1, are used to search the SMIP memory.

If a data unit with a matching key is found, it is deleted; if it is not found, no action is taken. At the end of the search, the ASC is not incremented. The need for the DFLAG arises out of the presence of negated keyword predicates in queries.

In Figure 25, we have reproduced from [3] an algorithm which performs an N-set intersection with appropriate comments in the light of our discussions.

4.2 Implementation Considerations

In carrying out set intersections, the most frequently used operation in the SMIP is the search-and-manipulate operation. Thus, it is obvious that in any implementation of the SMIP we must provide for an efficient and rapid search function. An important characteristic of this search function is that it always manipulates at most one SMIP data unit. This is because the data unit to be manipulated is identified by a unique key which forms a part of the data unit. This characteristic gives us a clue about a possible implementation technique, namely, hashing. Hashing as a search technique is effective only when the criterion for success is an exact match between a "search" key and a part of a data unit stored in the SMIP. Hashing is effective because it partitions the set of all possible search keys into equivalence classes by a relation called the hash function. Each of these partitions is implemented by a bucket memory. In order to locate a data unit with a key m, it is enough to search the bucket memory into which the key is hashed, instead of searching the entire SMIP memory. A system of bucket memories is implemented using the concept of PCAM that we used to implement the SM. In a PCAM-implemented SMIP each bucket is identified by a memory unit-processor pair. There are M such pairs. The intermediate set SW is partitioned into K subsets designated SWB_i, where $K \leq M$. The command SMIP<m,g> is executed by ordering all the K memory unit-processor pairs to retrieve valid data units. It should be noted that during the processing of an argument set, all the memory unit-processor pairs will be simultaneously manipulating data units corresponding to different search keys associated with different elements in the argument set. The size of the memory units and the number of pairs are dictated by performance requirements.

It may be interesting to point out some of the differences between the manner in which PCAMs are employed in the SM and the manner in which PCAMs are used in the SMIP. In the SM, a partition of the PCAM is implemented by allocating one or more modules in one or more memory units. In the SMIP a partition is usually completely contained in a single memory unit. [The case when

```
Argument Sets $X_1, X_2, ..., X_n$.
   begin
         for each $x_{1i}$ of $X_1$ do;
            begin
               execute the command  SMIP<create, $x_{1i}$>
Comment:  at this point ASC = 0, SMIP state = initial state;
            end
Comment:  ASC ← ASC + 1, thus SMIP state = active state;
         for j = 2, 3, 4, ..., N do;
            begin
               for each element $x_{ji}$ of $X_j$ do;
                   begin
                      execute SMIP<replace, $x_{ji}$>;
Comment:  ASC = j-1, state = active;
                   end
               ASC ← ASC + 1;
            end
   Execute the SMIP command  SMIP<retrieve>;
Comment:  ASC = n, state = retrieve;
```

Figure 25: A N-set Intersection Algorithm

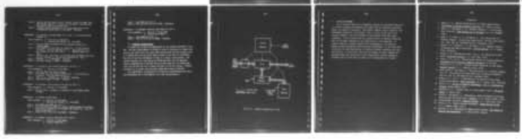
bucket overflow occurs will be discussed later as an exceptional case.] Because of the above difference, the type of parallelism available in the two components is different. In the SM, all PEs (except those that are masked) carry out the same operation on the same bucket (partition). In the SMIP, the same operation could be performed concurrently on several buckets (partitions), although all the operations would pertain to the same argument set. The memory unit-processing element pairs of the SM are synchronized in the execution of an order; this is not the case in the SMIP. Each memory unit-processor pair has a variable number of searches to be performed and are not synchronized. The variable number of searches is due to the fact that any hashing function, however good, will exhibit local nonrandomness leading to more searches in some buckets than in other buckets.

Why did we choose to implement PCAMs in two different ways in two components which apparently have the same major function, namely, searching? Before answering this question, we note parenthetically, that if we were given a choice, we would implement the SM PCAM in a manner similar to that of the SMIP PCAM, i.e. an implementation involving concurrent searching of several partitions. However, we cannot do this in the case of the SM because the SM search space is much larger than the SMIP search space. In fact it is easy to show that the entire SMIP search space is never greater than the largest partition of the SM. As a result of this size disparity, the technology applicable to the SMIP implementation is not applicable to the SM implementation. Whereas in the SMIP, we can use random access memories for each of the partitions, it is not cost-effective to use random access memories for the SM. Because of such a constraint, sequential or quasi-random access memories have to be used for the SM. The access times in such memories is a significant part of the overall retrieval time (i.e. access time + readout time). Now, a partition of SM is likely to be made up of several accessible units. Therefore, we can gain in performance by distributing these units over several of the processing elements and allowing them to concurrently access the units belonging to the same partition. Such a scheme will ensure that in the majority of instances the processing time will be no greater than one access time plus one readout time. Now, consider searching of several SM partitions concurrently. Since partitions may be contained entirely within the memory space attached to a single processing element, if there are two requests to two partitions in the same memory unit, then access to the partition corresponding to the second request will have to wait until the first request is completely process by the processing element. This may involve

AD-A035 178   OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 9/2
THE ARCHITECTURE OF A DATABASE COMPUTER. PART II. THE DESIGN OF--ETC(U)
OCT 76 D K HSIAO, K KANNAN   N00014-75-C-0573

UNCLASSIFIED   OSU-CISRC-TR-76-2   NL

2 OF 2
AD
A035178



END
DATE
FILMED
3-77

several sequential accesses to the memory unit attached to the processing element. Thus, we conclude, that for any sequence of requests that does not distribute uniformly over the partitions contained in the memory spaces of processing elements, concurrent processing of partitions consumes more time than concurrent processing of a single partition. In practice, we cannot ensure such uniformity of requests, and, therefore, it is advantageous to search a single partition at a time by a group of processing elements.

4.3 Physical Organization of the SMIP

The architecture of the SMIP is shown in Figure 26. The SMIP controller (SMIPC) manages the operation of the structure memory information processing elements (SMIPEs) via the SMIP control and data bus (SMIPCDB). Each SMIPE normally realizes one bucket of a hash table. The common memory bus (CMB) allows one SMIPE to use another's memory in case of bucket overflows in the former.

4.3.1 The SMIP Controller

The SMIP controller is responsible for the following functions:

- Maintaining the value of the argument set count (ASC)
- Hashing the elements of an argument set and determining the bucket to which the element belongs
- Controlling and issuing orders to the SMIP processing elements
- Interpreting the orders from the SM.

The maintenance of the ASC is an important function since it determines the command type. The ASC is an eight-bit counter which is set to zero or incremented by the algorithms described later in this section.

The hash function applied to the elements of the argument set in order to determine the bucket to be searched is simple and straightforward. Recall that the elements in the argument set are index terms retrieved by the BMS of the SM. Each index term is 28 bits long and is made up of 3 segments: an MAU address (8 bits), a cluster identifier (10 bits), and a security atom name (10 bits). The result of applying the hash function on this 28 bit index term is to produce an n bit number where 2^n is the number of MU - PE pairs in the SMIP. A simple hash function to do this would concatenate $[\frac{n}{3}]$ low order bits of the MAU address with $[\frac{n}{3}]$ low order

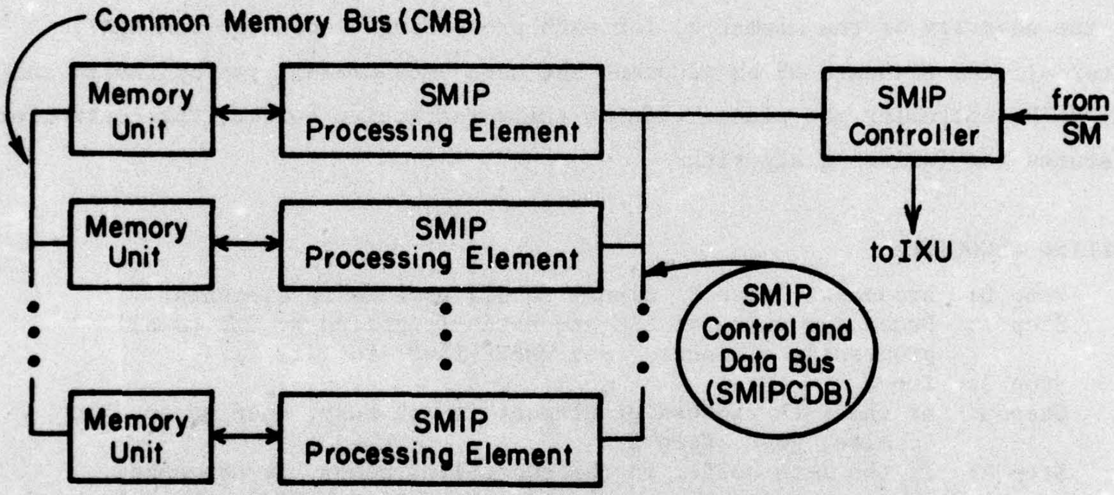


Figure 26. The Architecture of the SMIP

bits of the cluster name and $(n-[\frac{n}{3}] - [\frac{n}{3}])$ low order bits of the security atom name as shown in Figure 27. Of course there are several equally effective hash functions that could be used. The purpose of building one such function was to demonstrate the relative ease of creating hash functions for our application. Such function can be microprogrammed into the SMIP controller.

The n bit result of the hash function enables the controller to identify the bucket to be searched for a match. However, it is quite conceivable that the search time within a bucket is substantially longer than the time taken to produce the hash function. This observation implies that a speed mismatch is likely to develop between the controller and the processing elements. Therefore, the controller maintains a buffer (whose size depends on the severity of the mismatch) for each processing element in the SMIP. After all the elements of an argument set have been transmitted by the SM and the SMIP controller has placed them in their respective buffer, the controller executes the following algorithm

POLLING ALGORITHM

```
Step 0:  Broadcast 'search' signal to all processing elements.
Step 1:  Braodcast value of ASC and DFLAG (supplied by SM) to all
         processing elements.  Set $EMPTY[i]=0$ for all $i$.
Step 2:  $i \leftarrow 1$
Step 3:  If the $i$-th processing element is not busy, then go to Step
         4; else, go to Step 5
Step 4:  If the $i$-th buffer in the controller memory is non-empty,
         then transmit a search key from the $i$-th buffer to the $i$-th
         processing element.  If the $i$-th buffer is empty, set
         $EMPTY[i]$ to 1.
Step 5:  $i \leftarrow i + 1$.  If $i \leq N$, then go to Step 3.  If $i > N$ and
         $EMPTY[j]=1$, for all $j \leq N$, then go to Step 6; else, go
         to Step 2.
Step 6:  $ASC \leftarrow ASC + 1$.  Terminate.
```

Note: In the above algorithm EMPTY is an array of N bits where N is the number of processing elements in the SMIP. Each bit is used to indicate if the corresponding buffer is empty.

When the SM indicates to the SMIP controller that all the argument sets have been transmitted, the SMIP controller executes the following algorithm

RETRIEVAL ALGORITHM

```
Step 0:  Broadcast 'retrieve' signal to all processing elements.
Step 1:  Broadcast value of ASC to all processing elements.
Step 2:  Clear buffer memory in order to receive data units from the
         processing elements.
Step 3:  For all $i$, set $EODFLAG[i]$ to 0.
Step 4:  $i \leftarrow 1$.
```

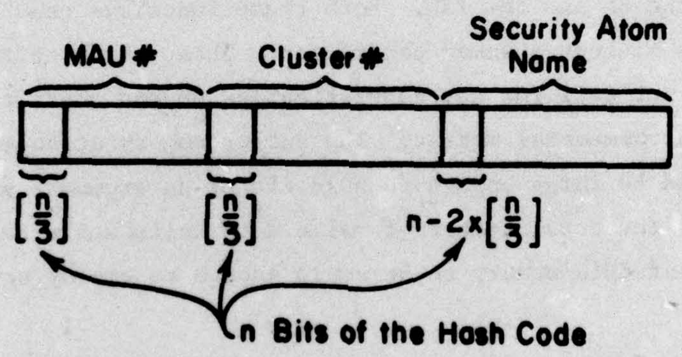

MAU #
Cluster #
Security Atom Name
$\left[\frac{n}{3}\right]$
$\left[\frac{n}{3}\right]$
$n-2\times\left[\frac{n}{3}\right]$
n Bits of the Hash Code

Figure 27. A Simple Hash Function

Step 5: If the i-th processing element has end-of-data flag on, go to Step 9.
Step 6: If the i-th processing element is ready to send a data unit, then go to Step 7; else, go to Step 8.
Step 7: Place the data unit from the i-th processing element in the buffer memory.
Step 8: $i \leftarrow i + 1$. If $i \leq N$ go to Step 5. If $i > N$ and if for all j EODFLAG[j]=1, then go to Step 10; else, go to Step 4.
Step 9: EODFLAG[i] ← 1. Go to Step 8.
Step 10: Transmit contents of buffer memory to IXU. Send 'clear' signal to all processing elements. Set ASC to 0 and clear buffer memory. Terminate.

Note: In the above algorithm EODFLAG is an array of N bits, each of which is used to indicate if the corresponding processing element has sent all the valid data units to the controller.

The SMIP controller is essentially a polling processor which monitors the activities of the processing elements. Its other important function is to interface with the SM and the IXU. Both these functions can be conveniently implemented into microprogrammed controller. Thus, the architecture of the controller does not call for any sophistication beyond what is already available in the commercial market. The buffer memory attached to the controller should be large enough to hold either an argument set (during intersection) or the total number of valid data units while retrieval. We expect the size of this memory to be small enough to employ semiconductor RAMs.

4.3.2 The Processing Element and Memory Unit Pairs

Each processing element has two main functions: First, it maintains the memory unit associated with it. By this we mean that the processing element is capable of adding, deleting, manipulating and retrieving data units in its memory in an efficient manner. Second, it interfaces with the SMIP controller. In this section, we shall address these two functions in some detail.

Of the four operations that need to be performed on the data units in the memory unit of the processing element, the most frequent operation is the manipulation function. The data unit to be manipulated, is identified by a search key supplied by the SMIP controller. Such identification is facilitated by once again employing a hashing strategy. The hash function employed within the memory unit of the processing element should not be confused with the hash function employed by the SMIP controller to identify the processing element-memory unit pair. The memory unit attached to a processing element is divided into two parts - a bucket index and a data memory. In this organization, all data units whose search key hashed to the same bucket name are placed in a linked list. The origin of the linked list is to be found in the bucket index, while the list itself is placed in the data unit memory, as illustrated in Figure 28. The hash function merely uses the high order n bits of the search key to index into the bucket index. The size of n depends on the relative size of the bucket index with respect to the size of the data memory. Each entry in the data memory has three fields: The first field has (28-n) bits of the search key; the second field is a count field; and the last field is a pointer to the next entry in the linked list. The first and second field together constitute a data unit.

In our earlier discussion in Section 4.1, we had indicated that the SMIP has three logical states. Corresponding to these three states, the processing element has three modes. These modes are set by the SMIP controller through control signals in the SMIPCDB. The three modes are called clear, search and retrieve modes. In the clear mode, the bucket index is cleared, thus effectively losing the data in the data memory. The processing element enters the clear mode at the receipt of a 'clear' signal from the SMIP controller. In the search mode, a search key is supplied by the SMIP controller along with the value of ASC and DFLAG. If the ASC value is zero, then an entry in the data memory is created for the search key with count field set to 1. If the value of ASC is greater than zero and if DFLAG is zero, then the search key is used to search the data memory for a matching entry. If it is found and if the count field equals the ASC value, then count field is incremented by one. If the matching entry is not found, or if the count field is not equal to the ASC value, no action is taken. If the value of ASC is greater than zero and if DFLAG is 1, then the matching entry (if found) in the data memory is deleted. The processing element enters the search mode upon receipt of a 'search' control signal from the SMIP controller. In the retrieve mode, the processing element systematically traverses the lists originating from the bucket index and transmits all those data units whose count field equals the ASC value supplied by the SMIP controller. The processing element enters the retrieve mode upon receipt of the 'retrieve' control signal.

There may be occassions when a memory unit associated with a processing element becomes full and cannot accomodate any more data units. Such a condition will occur only during the execution of the create function (ASC=0). When such a condition occurs, the concerned processing element

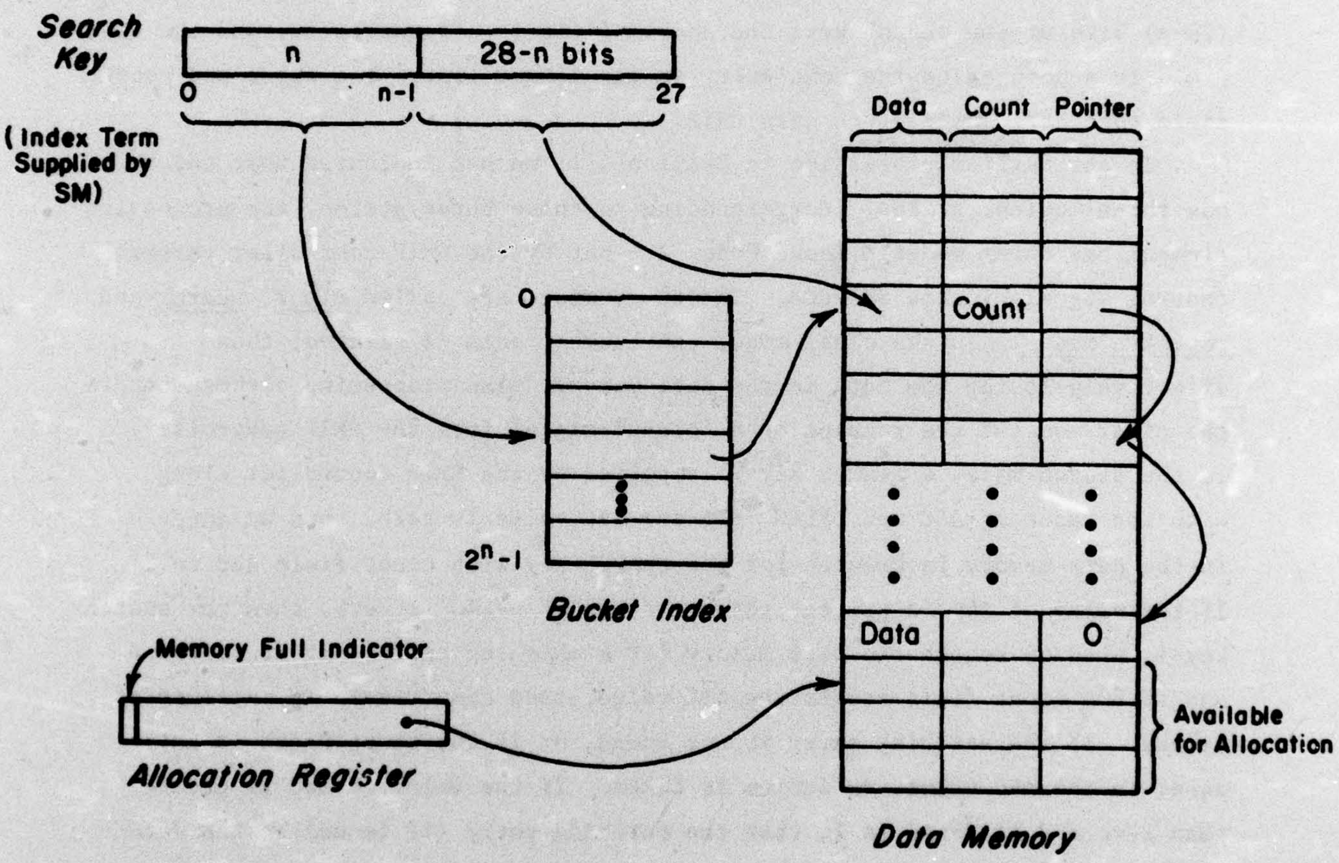


Figure 28. Data Structures Maintained by Processing Elements of SMIP

attempts to obtain space from some other memory units via the common memory bus. The common memory bus (CMB) shown in Figure 25 can be used to access the contents of any of the memory units. The prcoessing element which wishes to access its neighbor's memory unit must first obtain control of the bus. It can then address the memory unit of another processing element for reading or writing. All memory units have two I/O parts - one services requests from the processing element to which the memory unit is attached, and the second to service requests from the common bus. Before a processing element can use its neighbor's memory unit, it must make sure that space is available in the memory unit. This is done by examining the allocation register (see Figure 28). If the 'memory full' indicator is on, the the processing element has to try another memory unit for space. If the 'memory full' indicator is off, then the processing element reads off the contents of the allocation register and increments it by 1. The testing, reading and increment are carried out by a single indivisible instruction. Indivisibility of the above operation is important since, the local processing element should not be allowed to access or manipulate the allocation register while another processing element is accessing the same register via the CMB. Once the space for data unit has been allocated to a processing element, it can be manipulated in exactly the same manner as a local data unit. The processing element must remember the address of the memory unit from which it has borrowed the space. This is done by incorporating extra bits in the pointer field of a data unit in the data memory.

This completes our discussion on the logic of the processing element. As can be observed, the logic is fairly simple and can be implemented on a microprogrammed microprocessor. The memory requirements can be estimated only when we have an idea of the performance requirements of the SMIP. It is estimated that the requirements will be small enough to allow us to use random access memories for memory units attached to each of the processing elements.

5. THE INDEX TRANSLATION UNIT (IXU)

As a component of the DBC, the index translation unit (IXU) is responsible for translating an index term received from the SMIP into an absolute MAU address, a cluster identifier, and a security atom name. It is also responsible for allocating and releasing cluster identifier and security atom names as demanded by the DBCCP. In the ensuing discussion, we first present a rationale for having a separate unit to realize the above responsibilities. We next describe the data structures. We then describe the algorithms executed by the IXU. Finally, we estimate the hardware capabilities needed to realize the IXU.

5.1 The Need for an IXU

We learned that the KXU was specialized because of the need to produce transformed values of keywords efficiently. Also, the SM and SMIP hardware were designed to be efficient search engines. We may, perhaps, contemplate on incorporating the functions of the IXU among the three components already mentioned. Consider consolidating the functions of the IXU with those of the SM. We recall that the SM retrieves a large number of index terms which will ultimately be rejected by the SMIP because only a few index terms will survive the intersection operation. By prematurely translating all the index terms, the SM is performing work that is not really needed. Furthermore, by translating the index terms in the SM, we imply that SMIP must recognize and maintain three different data units corresponding to the three components of an index term instead of a single data unit. This in general would be reflected in more complex logic in SMIP. Thus, consolidating the IXU with the SM is not desirable. Next, consider consolidating the function of the IXU with the SMIP. In this case, in order to maintain concurrent intersection and translation operation a separate processor will have to be built into the SMIP to take care of the translation logic, while the SMIP controller is polling the processing elements of the SMIP for intersection. Such a decision essentially recognizes the independence of the IXU. Lastly we may consider consolidating the functions of the IXU with those of the DBCCP. We shall see in [8] that the DBCCP is primarily designed to achieve concurrency among separate components of the DBC. By making the IXU as a separate unit, additional concurrency can be achieved by the DBCCP. Thus, the four components, namely KXU, SM, SMIP and IXU, form a structure loop (see Figure 1) which serves as a pipeline of structural information of the database. We shall have more to say about this when we discuss the DBCCP in Part III.

5.2 Data Structures in the IXU

Each file known to the DBC (i.e., a file that is stored in the MM) has an MAU address table (MAUAT) maintained by the IXU. A MAUAT has up to 256 entries as shown in Figure 29. Each entry is two bytes long and contains the address of an MAU. A MAU address need be no longer than 16 bits, since the MM has less than 2^{16} MAUs. [This is true because we are designing a DBC to support a 10^{10} byte database and an MAU can typically hold no less than 2×10^5 bytes. This means there are no more than 50,000 MAUs in the MM]. Thus a MAUAT occupies 512 bytes. Associated with each MAUAT, is an MAU bit map (MAUBM) which indicates which (if any) of the 256 entries in the MAUAT is (are) free for allocation. The bit map occupies 32 bytes (= 256 bits). As we shall see in the next section, these two pieces of data structure are used in the translation of MAU numbers (in the index term) into absolute MAU addresses and in the maintenance of MAU numbers.

For each file, the IXU also maintains a cluster identifier bit map (CIBM) and a security atom name bit map (SANBM). These bit maps are used to keep track of the allocation and release of cluster identifiers and security atom names. Each bit map occupies 1024 bits to keep track of 2^{10} cluster names or security atom names. Thus the total space devoted to each file is given by 800 (= 512 + 32 + 128 + 128) bytes.

Under normal circumstances, we do not expect more than 2000 files to be existing within the DBC. Thus, the total memory capacity required is of the order of 1.6 M bytes. Fortunately, as we shall see in the next section, not all of the data structure need be accessible at any given time. In fact, since the index terms transmitted by the SMIP in any period of time corresponds to a particular file, it is enough if the IXU can have the data structure of that file in rapid access memory. The remaining data structures for other files may be in slower memory. We therfore propose that there be two types of memory associated with the IXU - a relatively small and fast memory (= 1000 bytes, implemented with semiconductor RAMs) for holding data structures of the current file and a large and slower memory (about 1.6 M bytes implemented with a fixed-head disk or equivalent) for holding data structures of other files.

5.3 Algorithms Executed by the IXU

The IXU is controlled by the DBCCP, and responds to commands from the DBCCP. The commands fall into two categories. The first category of commands requires input data (i.e., index terms) from the SMIP and results in the

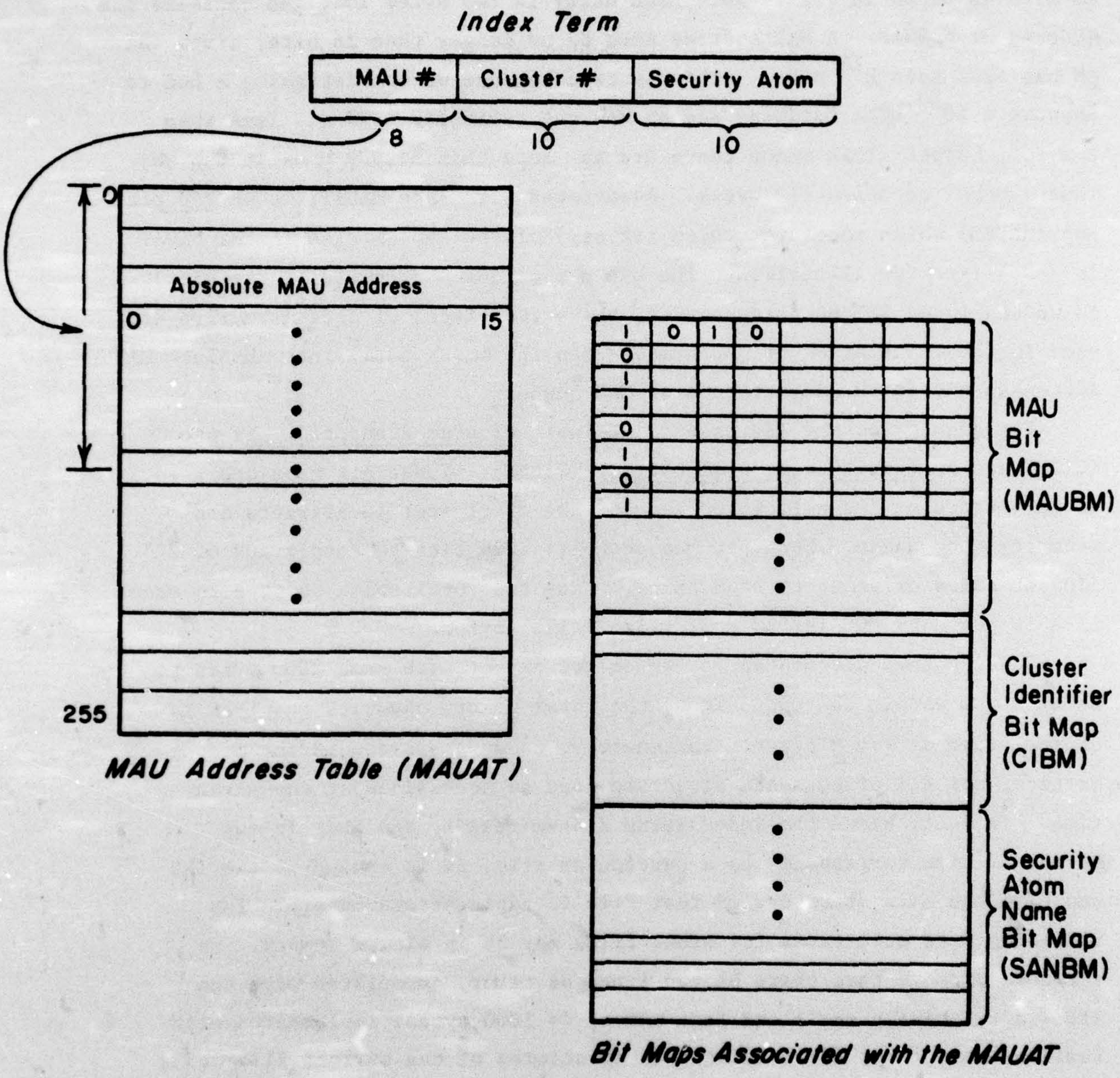

Index Term
MAU #
Cluster #
Security Atom
8
10
10
0
Absolute MAU Address
0
15
255
MAU Address Table (MAUAT)
1 0 1 0
0
1
1
0
1
0
1
MAU Bit Map (MAUBM)
Cluster Identifier Bit Map (CIBM)
Security Atom Name Bit Map (SANBM)
Bit Maps Associated with the MAUAT

Figure 29. Data Structures in IXU for a File

translation of the index term into MAU addresses, cluster identifiers and security atom numbers. The second category of commands is used by the DBCCP to allocate or release MAUs, cluster identifiers and security atom names. In the algorithms presented below, Algorithms A, B, C and D are executed in response to the first category of commands, while the remaining algorithms are executed in response to commands of the second category.

ALGORITHM A: Extract from the next set of index terms the MAU addresses with the help of MAUAT of file F.

Input Arguments: 1. Index Terms from SMIP
2. File ID, F, from DBCCP

Step 1: Load the MAUAT for the file F from the secondary memory into the primary memory.
Step 2: Wait for index terms from SMIP.
Step 3: For each index term from SMIP, do Step 4.
Step 4: Extract MAU number (high order 8 bits) from the index term. Use the MAU number as an index into the MAUAT. Retrieve the MAU address from the accessed entry.
Step 5: Transmit all MAU addresses to DBCCP. Terminate.

ALGORITHM B: Extract from the next set of index terms, the cluster identifier.

Input Arguments: 1. Index Terms from SMIP

Step 1: Wait for index term from SMIP.
Step 2: For each index term from SKIP, do Step 3.
Step 3: Extract the bits 8 through 17 of the index term.
Step 4: Transmit all the cluster identifiers extracted in Step 3 to DBCCP. Terminate.

ALGORITHM C: Extract from the next set of index term, the security atom names.

Input Arguments: 1. Index Terms from SMIP

Step 1: Wait for index terms from SMIP.
Step 2: For each term from SMIP, do Step 3.
Step 3: Extract the bits 18 through 27 of the index term.
Step 4: Transmit all the security atom names extracted in Step 3 to DBCCP. Terminate.

ALGORITHM D: Extract from the next set of index terms, the MAU addresses, the cluster identifiers and security atom names.

Input Arguments: 1. Index Terms from SMIP
2. File ID, F, from DBCCP

Step 1: Load the MAUAT from the file F from the secondary memory to the primary memory.
Step 2: Wait for the index term from SMIP.
Step 3: For each index term from SMIP, do Step 4.

Step 4: Extract the MAU number (bits 0 through 7) from the index term. Use the MAU number as an index into the MAUAT. Retrieve the MAU address from the accessed entry.

Step 5: Transmit the MAU address and the 20 low order bits of the corresponding index terms to the DBCCP. Terminate.

ALGORITHM E: To allocate a new MAU number for a file F and absolute MAU address M_a.

Input Arguments: 1. File ID, F, from DBCCP
2. Absolute MAU address M_a from DBCCP

Step 1: Load the MAUAT and MAUBM from the secondary memory into primary memory.

Step 2: Scan MAUBM for a bit which is turned off. If none found go to Step 5. Call the selected bit b_s. The position of b_s with respect to the first bit in MAUBM is given by $A(b_s)$.

Step 3: Turn b_s on. Use $A(b_s)$ as an index into the MAUAT, and place the MAU address M_a in the entry indicated by $A(b_s)$.

Step 4: Transmit b_s to the DBCCP. Terminate.

Step 5: Send error signal to DBCCP. Terminate.

ALGORITHM F: To allocate a new cluster identifier for file F.

Input Arguments: 1. File ID, F, from DBCCP

Step 1: Load the CIBM into primary memory.

Step 2: Scan CIBM for a bit which is turned off. If none found, go to Step 4. Call the selected bit b_s.

Step 3: Turn on b_s. Transmit b_s to DBCCP. Terminate.

Step 4: Send error signal to DBCCP. Terminate.

ALGORITHM G: To allocate a new security atom name for file F.

Input Arguments: 1. File ID, F, from DBCCP

Step 1: Execute Steps 1 through 4 of Algorithm F using SANBM instead of CIBM.

ALGORITHM H: To release an MAU number

Input Argument: 1. File ID, F, from DBCCP
2. Absolute MAU address M_a from DBCCP

Step 1: Load MAUAT and MAUBM for file F.

Step 2: Scan the entries of MAUAT to obtain a match between the contents of an entry in MAUAT and the argument MAU address M_a. Call the index of the entry k.

Step 3: Turn off the k-th bit in the MAUBM. Terminate.

ALGORITHM I: To release a cluster identifier for a file F

Input Arguments: 1. File ID, F, from DBCCP
2. Cluster Identifier k

Step 1: Load CIBM for file F.
Step 2: Turn off the k-th bit in CIBM. Terminate.

ALGORITHM J: To release a security atom name for file F

Input Arguments: 1. File ID, F, from DBCCP
2. Security atom name k

Step 1: Load SANBM for file F.
Step 2: Turn off k-th bit in SANBM. Terminate

5.4 Hardware Considerations

In Figure 30, we show the organisation of the various components of the IXU. The fast access memory is limited to what is required by a single file, since at any given time, the index terms ariving from the SMIP pertains only to a single file, and the file's name is known to the IXU (via the DBCCP). The bulk of the file information is kept on a larger, slower memory. The leading candidate for this memory are fixed-head disk and CCD memories. Certain organizations of CCD have better access times than the fixed head disk, and the cost per bit is almost the same for the two devices. If CCD's volatility is at issue, then the use of disk may be more attractive.

The algorithms to be executed by the IXU are straightforward and can be microprogrammed into a minicomputer (or a fast microprocessor).

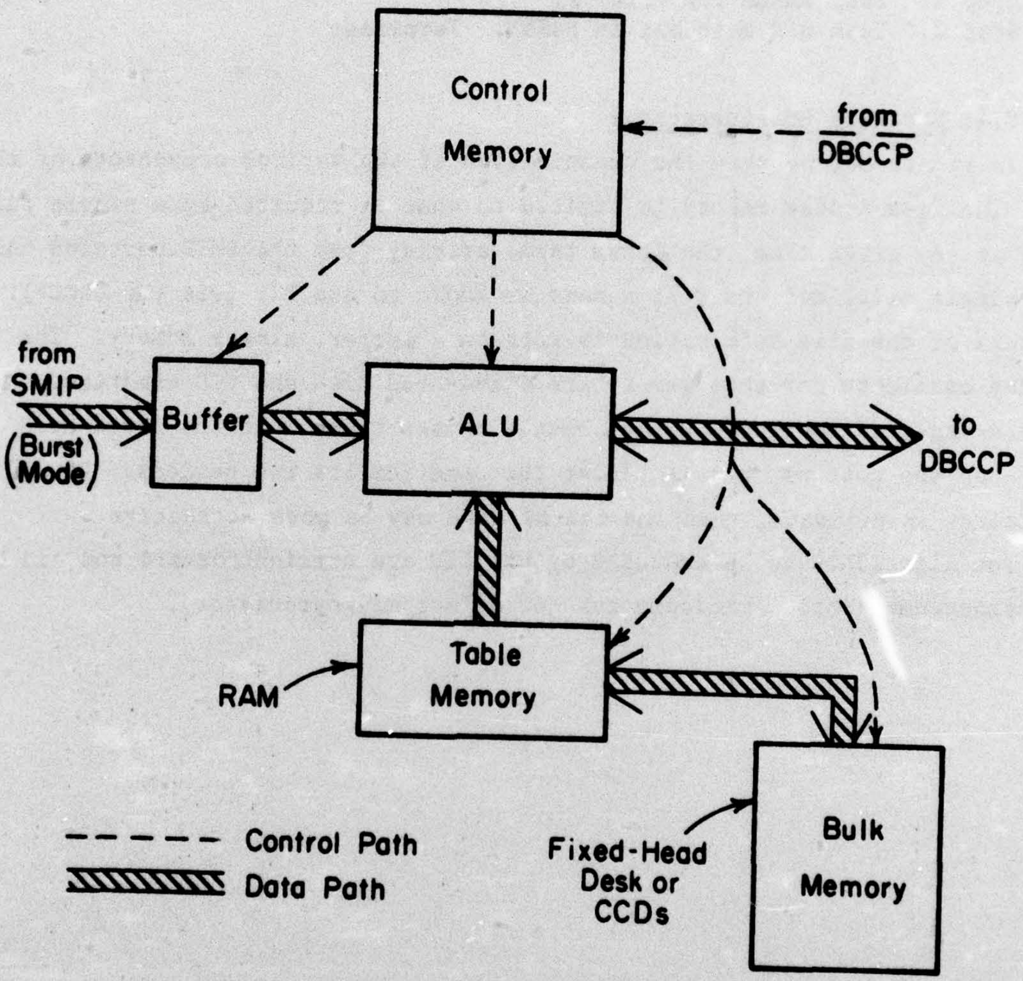


Figure 30. Hardware Organization of IXU

6. CONCLUDING REMARKS

One of the main reasons for presenting a design in considerable detail is to convince ourselves and hopefully the reader that the components described herein can indeed be contructed. The details, however, do not include a cost-performance evaluation of the design. This, in fact, is the topic of an on-going research. The SM was treated in much greater depth than other components. This is because that the characteristics of emerging technology have the strongest influence on the capabilities of this component. It became necessary to expound carefully the requirements of the SM and to examine design alternatives using different technologies. On the other hand, we have not included descriptions of interfaces between components and their bus protocols. We felt that, with the given design of simple commands and straightforward data structures for the communication between components, interface and bus structures are likely to be conventional and routine.

References

1. Amelio, G. F., "Charge-Coupled Devices for Memory Applications", Proceedings of National Computer Conference, 44, (1975), 515-522.
2. Baum, R. I. and Hsiao, D. K., "Database Computers - A Step Towards Data Utilities", IEEE Transactions on Computers, C25, 12, (December 1976).
3. Baum, R. I., Hsiao, D. K., and Kannan, K., "The Architecture of a Database Computer - Concepts and Capabilities", The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-76-1, (September, 1976).
4. Bobech, A. H., Bonyhard, P. I. and Guesic, J. E., "Magnetic Bubbles - An Emerging New Memory Technology", IEEE Proceedings, 63, 8, (August 1975), 1176-1195.
5. Bowers, D. M., "Systems-on-a Chip - Part 1: The Revolution in Full Power", Mini Micro Systems, 9, 5, (May 1976), 74-76.
6. Carnes, J. E., et al., "Charge-Coupled Devices for Computer Memories", Proceedings of 1974 National Computer Conference, 43, 827-836.
7. Coulouris, G. F., et al., "Towards Content Addressing in Data Bases", Computer Journal, 15, 2, (February 1972), 95-98.
8. Hsiao, David K. and Kannan, K., "The Architecture of a Database Computer Part III: The Design of the Mass Memory and its Related Components", OSU Tech. Rep. No. OSU-CISRC-TR-76-3, (December 1976).
9. Hughes, W. C., et al., "A Semiconductor Non-volatile Election Beam Accessed Mass Memory", IEEE Proceedings, 63, 8, (August 1975), 1230-1240.
10. Knuth, D. E., The Art of Computer Programming - Vol. 3 - Searching and Sorting, Addison Wesley, 1973.
11. Kosonocky, W. F., "Condiser CCDs for a Wide Range of Uses", Electronic Design, 6, (March 1976).
12. Levine, L. and Meyers, W., "Semiconductor Memory Reliability with Error Detecting and Correcting Codes", Computer, 9, 10, (October 1976), 43-50.
13. Microprocessor Scorecard, Minimicro Systems, 9, 7, (July 1976), 44-45.
14. Morris, R. H., "Scatter Storage Techniques", Communications of ACM, 11, 1, (January 1968), 38-43.
15. Peterson, W. W., Addressing for Random Access Storage", IBM Journal of Research and Development, 1, 2, (April 1957), 130-146.